DIALABLE CRYPTOGRAPHY FOR WIRELESS NETWORKS

THESIS

Marnita Thompson Eaddie, Major, USAF

AFIT/GCO/ENG/08-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

AFIT/GCO/ENG/08-02

DIALABLE CRYPTOGRAPHY FOR WIRELESS NETWORKS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Marnita Thompson Eaddie

Major, USAF

March 2008

AFIT/GCO/ENG/08-02

DIALABLE CRYPTOGRAPHY FOR WIRELESS NETWORKS

Marnita Thompson Eaddie
Major, USAF

Approved:

// signed //	March 6, 2008
Dr. Kenneth M. Hopkinson, PhD (Chairman)	Date
// signed //	March 6, 2008
Lt Col Stuart H. Kurkowski, PhD, USAF (Member)	Date
// signed //	March 6, 2008
Capt Ryan W. Thomas, PhD, USAF (Member)	Date

Abstract

The objective of this research is to develop an adaptive cryptographic protocol, which allows users to select an optimal cryptographic strength and algorithm based upon the hardware and bandwidth available and allows users to reason about the level of security versus the system throughput. In this constantly technically-improving society, the ability to communicate via wireless technology provides an avenue for delivering information at anytime nearly anywhere. Sensitive or classified information can be transferred wirelessly across unsecured channels by using cryptographic algorithms. The research presented will focus on dynamically selecting optimal cryptographic algorithms and cryptographic strengths based upon the hardware and bandwidth available. The research will explore the performance of transferring information using various cryptographic algorithms and strengths using different CPU and bandwidths on various sized packets or files.

This research will provide a foundation for dynamically selecting cryptographic algorithms and key sizes. The conclusion of the research provides a selection process for users to determine the best cryptographic algorithms and strengths to send desired information without waiting for information security personnel to determine the required method for transferring. This capability will be an important stepping stone towards the military's vision of future Net-Centric Warfare capabilities.

Dedication

This work is dedicated to my husband, my four wonderful children, and to my mother. Their steadfast love and support never wavered; and they always believed that I could achieve my goal.

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Dr. Kenneth Hopkinson, for his guidance and support throughout the course of this thesis effort. His insight and experience into my thesis topic was highly beneficial. Special thanks as well to Matt Weeks for the initial writing of the gpgTester code used in this research. I also thank my sponsor, Mr. Robert Bonneau, from the Air Force Office of Scientific Research for both the support and latitude provided to me in this endeavor.

Marnita Thompson Eaddie

.

Quote

"If plans related to secret operations are prematurely divulged the agent and all those to whom he spoke of them shall be put to death."

Sun Tzu
Art of War page 147, no 15

Table of Contents

Page

Page

List of Figures

Page

Page

List of Tables

Page

Acronyms

3DES:	Triple Data Encryption Standard
AES:	Advanced Encryption Standard
ASM:	Adaptive Security Model
ASPAD:	Adaptive Quality of Security Control Scheme
DRE:	Distributed Real-time Embedded
DSOCARE:	Dynamic Selector of Optimal Cryptographic Algorithms in a Runtime Environment
ELG-E:	ElGamal
GPG:	GNU Privacy Guard
HANC:	Hybrid Agent for Network Control
MANET:	Mobile Ad-hoc Network
NetA:	Network Attack
NetD:	Network Defense
PI:	Performance Index
ROCAS:	Runtime Optimal Cryptographic Algorithm Selector
RSA:	Rivest, Shamir, and Adleman
RTES:	Real Time Embedded System
SI:	Security Index
SoD:	Strength of Defense
SPI:	Service Provision Index

SSTT: Systematic Security and Timeliness Tradeoffs

WAHSN: Wireless Ad-Hoc and Sensor Network

DIALABLE CRYPTOGRAPHY FOR WIRELESS NETWORKS

I. Introduction

1.1 Motivation

Cryptography continues to play a major role in the military and in the public sector. The utilization of cryptography has existed for centuries. Early Egyptians used hieroglyphics on tombs as riddles or puzzles for visitors to decipher. Around 1500 B.C., scribes from Mesopotamia used encrypted cuneiform tablets for keeping pottery glaze formulas secret. The first known use of cryptography for military communication was in 475 B.C when the Spartans used the transposition cipher (Mollin, 2005).

Military communications continue to use forms of cryptography. Examples include the substitution cipher used by Julius Caesar, the wheel cipher invented by Thomas Jefferson, the Playfair cipher used by the British Foreign Office in the 1800s, forms of Vigenere ciphers used by the Confederate army, and the Enigma machine used by the Germans during World War II (Mollin, 2005).

From ancient history to present day, cryptography has enabled people to communicate secretly with some confidence in the chosen cryptographic scheme to assume that adversaries will be unable to interpret the communication in a timely manner. The basic need for cryptography has not changed since its integration with technology but the schemes have evolved as we become more complex and demand more complex

technology. Static security methods and statically chosen cryptographic schemes cannot adjust to changing environmental factors especially in the wireless environment. To meet dynamic environmental security requirements an adaptive approach is necessary. Adaptive security using cryptography is one method to affect the security posture of dynamic environments.

1.2 Overview of Adaptive Security

Adaptive security is a collection of traditional security measures, vulnerability monitoring, detection, and response. Adaptive security can be manual or automatic although a manual method does not allow for responding quickly in a fast changing environment. Automatic methods delve into the arena of evolutionary computing where patterns in the biological world are observed and modeled including dynamic learning, dynamic decision-making, self-guidance, and self-repair. Evolutionary computing should be able to handle the future optimization and security challenges that arise with increased technology including the distribution, scale (micro to massive), autonomy, and mobility of systems.

Using adaptive security, a system can exist in a less secure but higher performing state for normal operation and then can adapt to a more secure and usually less performing state when negative factors arise within or outside the system (Hinton, unpublished). Because it is adaptive, the system can utilize different cryptographic algorithms.

With adaptive security, a fine balance must be struck between security (cryptographic computations) and performance, or, for distributed real-time embedded

(DRE), between real-time monitoring and cryptographic computations competing for computational time and resources. This inherent overhead from cryptographic computations can strain the infrastructure. In addition, the security schemes used can also affect the system throughput and latencies that can potentially make networked applications more vulnerable to attacks or denial of services. Research is under way to solve various adaptive security problems involving runtime environments, mobile ad-hoc networks, networked parallel disk systems, and real-time embedded systems utilizing various schemes (e.g. Alampalayam, 2003; Soliman, 2005; Raissi, 2006; and Nijim, 2006). The research in this thesis focuses on allowing users to initiate 'dialable' cryptography for dynamic networks including wireless networks.

The idea of dynamically changing the security of a system is important in wireless ad hoc and sensor networks where critical resources such as battery life, memory, computational power, and bandwidth are not constant nor necessarily predictable. Mobile ad hoc networks are also susceptible to intrusion, eavesdropping, and selfishness (where one node may demand and receive more resources than another) (Chigan, 2004). For these vulnerable wireless enabled networks or systems, confidentiality, integrity, and authentication is critical. If the systems include distributed real-time embedded (DRE) applications it becomes even more crucial to verify that adversaries have not read data (confidentiality), have not modified data (integrity), nor claim false identity (authentication). These systems or applications rely heavily on cryptographic schemes for information processing. DREs need to report the real world status to the proper personnel. Examples of DREs include electric grid management (report on electric grid

status for determining energy supply plans) and defense applications (report on the battlefield to military control center for preparing overall battle tactics).

1.3 Military Requirement for Wireless Technology

Cryptography is an important player in wireless technology which in turn plays a crucial role in the military. "One would be hard-pressed to name a wireless technology that didn't have its beginnings in the military world" (Blyler, 2004). The military is, of course, concerned with information operations.

A subset of information operations is Computer Network Defense. "NetD operations protect and defend friendly information systems, computer networks, and information transiting within them. In addition, they protect against the NetA (Computer Network Attack) capabilities of others" (Franz, 2007). Presently, network personnel concentrate NetD efforts predominantly on the NIPRNET. However, non-wired technology is increasing exponentially. "Military worldwide are moving toward the concept of network-centric operations: networking their forces with wireless communications technologies to increase combat effectiveness" (Pucker, 2007). Bandwidth intensive applications like video teleconferencing, secure telephony, and imagery transfers are now requirements we expect in military communications and networks to increase situational awareness. These networks allow easy installation, lower installation costs, and flexibility (no traditional wired physical restrictions).

Wireless technology can be used by the military for sensors and sensor networks, unmanned vehicles, aircraft, communications, etc. With usage, however, comes the responsibility of securing the technology against compromise. Current security

techniques are concerned with meeting requirements for three security principles: confidentiality, integrity, and authenticity, principles which can be fulfilled via cryptography.

Without strong, reliable cryptographic algorithms, the data collected via different technologies would not be trustworthy and the information collected would be too precarious for military decision makers. In fact, because unencrypted wireless traffic is susceptible to sniffing, the Department of Defense published Directive 8100.2 in April 2004 to address the emerging new technologies. The directive mandated encryption on unclassified commercial wireless devices, active screening for wireless devices (unknown attackers could set up wireless devices at perimeters), and incorporation of wireless information into annual Information Assurance training (military personnel should not be the weak link because of a lack of training) (Defense, 2004).

1.4 Overview of Research

The objective of this research is to develop an adaptive cryptographic controller, which allows users to reason about the level of security versus the system throughput.

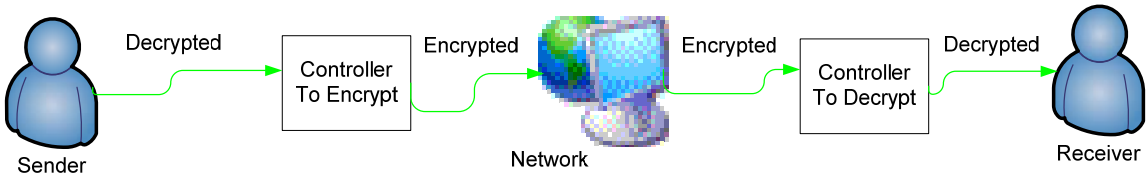

Figure 1: Thesis Objective

The ability to communicate via wireless technology provides an avenue for delivering information at anytime. Sensitive or classified information can be transferred wirelessly across unsecured channels by using cryptographic algorithms. My research

focused on dynamically selecting optimal cryptographic algorithms and cryptographic strengths based upon the hardware and bandwidth that were available for the transferring of various sized files or packets. Simulations were used to test the controller that was developed in my research.

The conclusion of the research provides a selection process for users to determine the best cryptographic algorithms and strengths to send desired information without waiting for information security personnel to determine the required method for transferring. This capability will be an important stepping stone towards the military's vision of future Net-Centric Warfare capabilities.

II. Literature Review

2.1 Underlying Theme

Research in cryptography, to meet the growing demands of wireless technology, continues to expand. Some of the research focuses closely on the research objective, adapting cryptographic algorithms based upon the observed wireless network environment, i.e. a dialable cryptography for wireless networks. The different nuances for the term 'dialable cryptography' include dynamic cryptography or adaptive security. The literature review of this research was used in the research methodology.

The basic premise of different on-going dynamic cryptography is that if cryptographic algorithms are predetermined before programs are run or data transferred, then the algorithms are based upon assumptions agreed upon prior to runtime (based on earlier behavior) and does not take into consideration the actual runtime environment. These algorithms are static, predictable, inefficient, and bind the user to security decisions made at the beginning of the communication session or network connection (Raissi, 2006). Monitoring is required to determine if the static algorithm is sufficient for the task. The optimal approach is to dynamically select the cryptographic algorithm at runtime, based upon the state of the actual system, to balance the tradeoffs between security, speed, and priority for the files to be transferred.

2.2 DSOCARE

A model to do this is the Dynamic Selection of Optimal Cryptographic Algorithms in a Runtime Environment (DSOCARE). DSOCARE worked at the middleware level to collect, select, and report on symmetric block ciphers.

The DSOCARE's collection function included running benchmark tests and storing the data in database management systems located on the DSOCARE server. (Raissi 2006) The model used a Runtime Optimal Cryptographic Algorithm Selector (ROCAS) that was based on a hybrid genetic algorithm (part of evolutionary computing where biological systems are observed). To select optimal algorithms, ROCAS received diverse client security requests with high and low security, speed, and prioritized quality of service parameters. The reporting function of DSOCARE was used to generate and gather data.

DSOCARE's basic approach is that one algorithm does not fit all situations. The centralized approach (middleware) allowed for the collecting, analyzing, and recommending optimal cryptographic algorithms. Since communication services vary in security, speed, and priority, the chosen optimal cryptographic algorithm was based on the output from the genetic algorithm (which included client input) and not a statically predetermined one.

The results from DSOCARE's data indicate that security, speed, and priority are improved with running DSOCARE as opposed to traditional cryptographic static services or methods.

2.3 MANET Frameworks/Models

A divergence from the standard delivery of cryptographic methods is on-demand key distribution for mobile ad-hoc networks (MANETs). This method uses message keys that are passed through the network to avoid encrypting and decrypting messages at every node. Instead the message key is the only item re-encrypted.

MANETs are subject to intermittent connections caused by obstacles blocking transmission or other interference. In MANETs, there may be a need to have secure multicast communications where many nodes receive the encrypted message at once instead of unicast where an encrypted message is designated for one node. Because of the unreliability of MANETs and multicast messaging, on-demand key distribution becomes a necessary method for communication. This distribution uses gossip-based protocols which use epidemic communication patterns (these patterns are observed in biological systems when diseases are spread). Like the spread of a disease, there are healthy nodes (reliable) and unhealthy nodes (unreliable). The healthy nodes will receive the message in a logarithmic number of rounds while the unreliable nodes will receive it at an intermittent rate (Graham, Hopkinson et al. 2007).

Although this thesis did not focus on the actual distribution of the cryptographic keys, the research by Graham provided a background investigation into the key distribution challenges in wireless networks. In addition, it emphasized measuring end-to-end delay from prior to encryption to after encryption. Another observation from the paper is that symmetric key algorithms are usually faster than public key although "in practice, … symmetric key encryption is preferred, but public key encryption is often

required in order to send a symmetric key for future use" (Graham, Hopkinson et al. 2007).

Another model for MANETS is the Adaptive Security Model (ASM) for denial of service security threats in mobile agent systems (Alampalayam and Kumar 2003). It is based on a fuzzy feedback controller. Fuzzy feedback controllers are used with fuzzy logic which deals with approximate reasoning similar to how humans logically arrive at answers given imprecise information. The user selects security levels and requirements that he/she needs. Since MANETs are more susceptible to intrusion, eavesdropping, and selfishness, the "goal is to provide a security framework that will detect automatically various attacks and take appropriate measures to minimize the denial of service effects" (Alampalayam and Kumar 2003). This flexible and adaptable framework created by the model is scalable, and could detect, prevent, and control security attacks at node-level or system-level (Alampalayam and Kumar 2003). The ASM integrates security requirements by combining holistic ("holistic security is proactive, preventive, and predictive" (Alampalayam and Kumar 2003)) and adaptive security techniques to allow users who are not experts in security to easily decide security policies. The ASM uses a feedback control scheme similar to how a human reacts to a virus (evolutionary computing) and assigns security levels (Alampalayam and Kumar 2003).

The concepts of how the user selects security levels and requirements provide background information for this research and were incorporated into the methodology.

2.4 WAHSN

Similar to Mobile Ad-Hoc Networks, Wireless Ad-hoc and Sensor Networks (WAHSN) can utilize adaptive security. Critical resources ("battery life, memory, computation, power, and bandwidth" (Chigan, Ye et al. 2004) for WAHSNs are used as parameters for a resource-aware security framework. This self-adaptive framework takes into consideration performance cost and network resource expenses. The basis of the framework is a knowledge profile that includes system vulnerabilities, system security requirements, system network performance requirements, and critical resources. The security attributes are quantified into a security index (SI). Each specific WAHSN application (for example integrity or confidentiality) is given a value on a scale with associated resource costs. The SI quantifies how secure the system is. Another quantification is the Performance Index (PI) which quantifies the network performance. The Service Provision Index (SPI) combines security and performance. Ye Chigan's research was investigating two different optimization modules for security. Both of the modules should determine the best cryptographic scheme to apply to the WAHSN system under different communication protocols and resource constraints. Important points from Chigan's research which was applied to this research are that the mechanism (controller) must take into consideration various limiting factors (i.e., it cannot just choose the highest level of cryptographic algorithms) and that the controller seeks to maximize the overall network security service and network performance service. In addition, it can switch from one protocol (including the cryptographic algorithms) under attack to another protocol (Chigan, Ye et al. 2004).

2.5 DRE

In addition to MANETs and WAHSNs, distributed real-time embedded (DRE) applications use wireless technology. For DREs, the criticality to report in a timely manner is countered by the threat of adversaries reading data, modifying it, or claiming false identity (Kang and Son, 2006). The model, Systematic Security and Timeliness Tradeoffs (SSTT), balances time and cryptographic security requirements in applications such as battlefield monitoring and target tracking. The model introduces the variable, Strength of Defense (SoD), based on the cryptographic key length. The cryptographic key length is adapted to improve the performance of DREs. The model does not test for dynamic key generation or distribution but assumes that the keys are distributed and agreed upon prior to running the programs and only symmetric keys are used.

The model attempts to achieve three main goals: confidentiality, integrity, and authenticity. Confidentiality is supported when a Real Time Embedded System (RTES) encrypts the plaintext message and prevents replay attacks by using counters. The integrity and authenticity of the message is achieved by creating a secure message authentication code that is computed over the message using a secure one-way hash function. There is an exhibited tradeoff between the cryptographic algorithms and speed. For example, if an RTES inside of an unmanned aerial vehicle normally use AES-256, it can decrease to AES-128 when it is overloaded and unable to perform well. The SSTT algorithm is not tied to a specific cryptographic algorithm nor key length (Kang and Son, 2006).

2.6 Parallel Disk Systems Model

Mobile Ad-hoc Networks, Wireless Ad-hoc and Sensor Networks, and Distributed Real-Time Embedded applications are not the only networks/applications that benefit from adaptive security. Parallel disk systems can also benefit. Although they can alleviate disk input/output bottleneck problems, these highly scalable systems normally do not allow the optimal and dynamic changing of the networked environment.

A model currently under research for parallel disk systems is the Adaptive Quality of Security Control Scheme (ASPAD). It allows network disk systems to adapt to changing security requirements and workload conditions. ASPAD has three phases: "dynamic data partitioning, response time estimation, and adaptive security quality control" (Nijim, Qin et al. 2006). Through the successful completion of the phases, ASPAD can achieve two major performance goals: high quality of security and guaranteed response time.

The ASPAD research methodologies was used as background for the methodology because the ASPAD determined the cryptographic scheme for the disk requests while still trying to achieve two performance goals: high quality of security and guaranteed response time.

2.7 AdaptCrypt

Another aspect of choosing cryptographic algorithms for adaptive security is incorporating the security policies into the decision-making process. Security policies take into consideration system assets (cryptographic algorithms) and security responsibilities. One method concerns flexible encryption for files where the entire file

or text may have different types of encryption. "The AdaptCrypt model uses several different encryption algorithms (block ciphers for example AES, BlowFish, and IDEA) to encrypt a file" (Manzanares, Camara et al. 2005). The security levels can be modified by changing the encryption algorithms or switching key lengths for chosen parts of the file. For example, one pattern for a four-block file could be AES 256-Blowfish-AES-128-TwoFish. If the encryption pattern changes, only the portion that is incorrect would need to be modified. Using parallelized encryption, each of the encrypted blocks can be individually accessed. ApaptCrypt is not automatic but relies upon the user to manually encrypt and decrypt based upon the security policies. Further testing of AdaptCrypt could lead to enhancements for automatically choosing the security levels (cryptographic algorithms) based upon the outcomes of the security policies.

2.9 Context-Based Security

Another method for adaptive security via security policies is with context-based security. Context-based security considers the context in which the system is used. It is a relatively new approach to counter different types of security problems brought on by "increased mobility of pervasive systems and heterogeneity of devices" (Brezillon, 2004). In these systems, the context continually changed based upon dynamic environmental variations and was used to determine the security context. The security context is a set of collected information from the user and application environment that influences the security infrastructure of the user and the application (Brezillon, 2004). This security context describes situations in which security decisions were made. These decisions can include adapting the cryptographic protocol used, requiring stronger authentication, or

automatically denying access when intrusion was detected. The model used contextual graphs which were "chance nodes where contextual elements are analyzed to select the corresponding path." (Brezillon, 2004). The graphs take into account the actual working environment.

2.10 SAM

Although the research by Hinton on a Security Adaptive Manager, SAM, refers mainly to operating systems under different conditions and not necessarily with cryptography adaptations, there were interesting ideas that was utilized in this thesis.

First "using adaptive security, a system can exist in a less secure, more performant state until it comes under attack" (Hinton, Cowan et al., unpublished).

Second, "adaptive security allows us to implement a system in a high performance, highly functional state for normal use, and then adapt the system to a less performant/less functional/more secure state in the presence of attacks" (Hinton, Cowan et al., unpublished).

Third, it discusses an adaptation space which defines when and how adaptation is implemented. The adaptation space includes condition space and transition graph. The condition space is a complete lattice of possible conditions of interest and their settings. In the transition graph, there is one node for each system configuration. The graph defines how to transition between different configurations (implementation alternatives).

III. Methodology

3.1 Methodology Overview

This thesis investigated the development of software controllers that would choose an encryption scheme for a commodity or a list of commodities. These commodities represented the information to be sent at a node in a wired or wireless network. At the node, the commodities will be encrypted (simulated) and then transmitted over the network, however with limited bandwidth and CPU, only a select list of commodities can be transmitted. Since each commodity consisted of a file size and a priority for the file size, the list of transmitted commodities were the ones that maximized the sum of the priorities (i.e. the goodness) and still maintained the priority for the commodity (e.g. a priority of 80 would be transmitted before a priority of 20 if possible).

In making the decision for an encryption scheme, the controller had to factor in the available bandwidth and CPU for the entire list of commodities sent to it. In addition, the performance characteristics (i.e. the security level) of different encryption schemes determined the transmitted commodity list.

The end results of the thesis were four different controllers developed which allowed two different methods for choosing the encryption scheme (via binary integer programming and without binary integer programming). Figure 2 shows a graphical representation of the research objective.

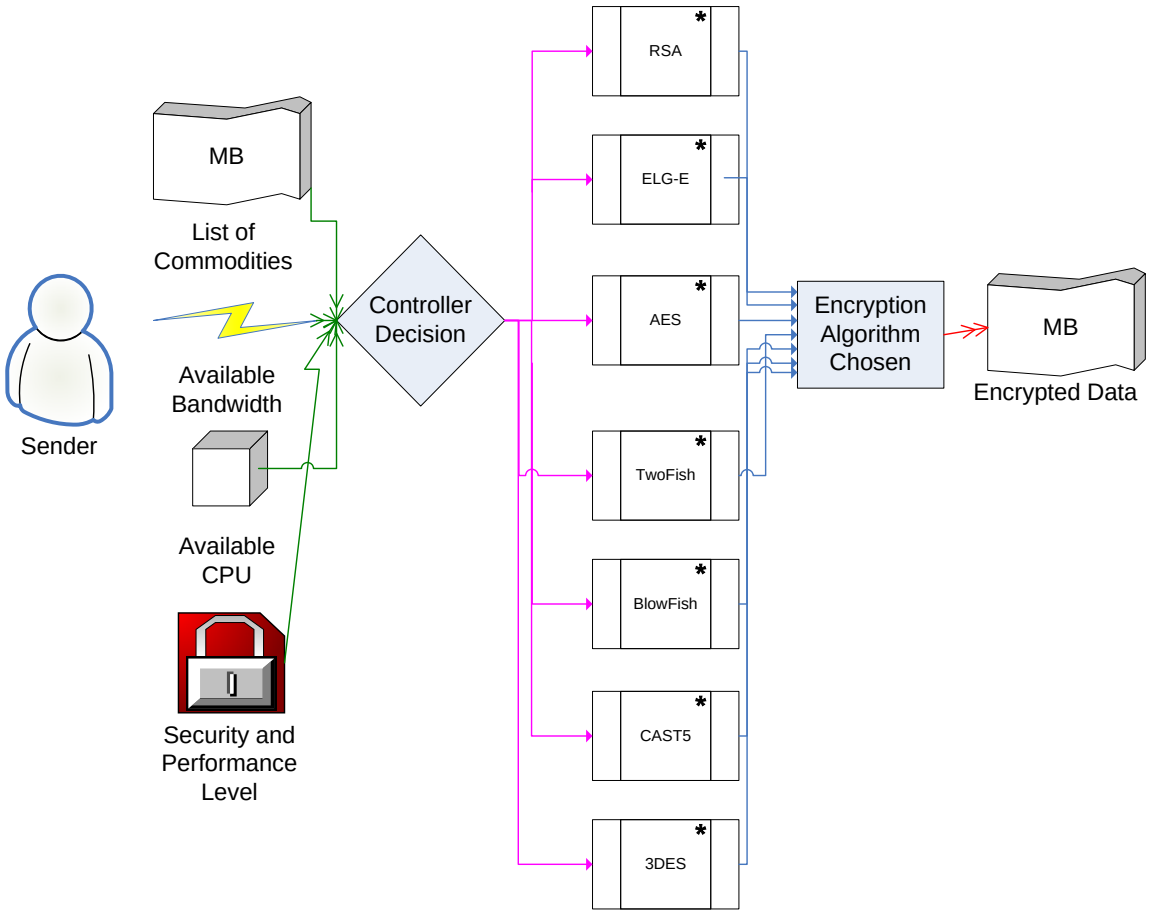

Figure 2. Detailed Thesis Objective

The controllers could choose between seven encryption algorithms (3DES, AES, BlowFish, CAST5, ElGamal, RSA, and TwoFish) and various key sizes.

To meet the research goal of creating controllers for selecting encryption schemes, the research was divided into four steps. Step 1 developed a front end (gpgTester) to the cryptographic tool, gpg, GNU Privacy Guard. The front end, gpgTester, was used to time how long various cryptographic algorithms took to encrypt and decrypt various sized files. Step 2 inserted the data from Step 1 into MatLab to interpolate the given inputted data based upon encryption schemes. Step 3 created the

controller by optimizing the interpolated data from Step 2 utilizing MatLab. Step 4 used NS-2 to run simulations for the controller. The flowchart in Figure 3 depicts the four major steps in the research.

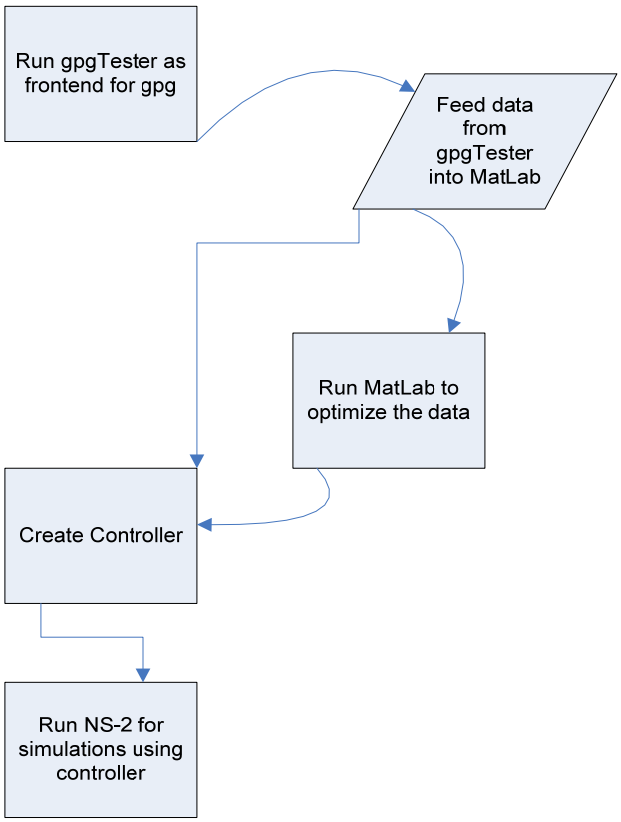

Figure 3: Overall Methodology Flowchart

The primary tools used in this research were C++, gpgTester (code written for this research), gpg (a tool for using different cryptographic algorithms), MatLab (used to interpolate the data, optimize the computations from gpgTester data, and create the controller), and NS-2 (used to run simulations). The simulations and controllers were based upon the results gathered from the gpgTester program.

3.2 gpgTester (Step One)

3.2.1 gpgTester Explanation.

The first step of the research was to actually collect statistical data from different encryption algorithms and key sizes. This data, the foundation for the other research steps, included the encryption time, encrypted file size, and the decrypted time for different files.

To collect the data, the open source cryptographic tool, gpg (GNU Privacy Guard), was used. This tool encrypted and decrypted files using various encryption algorithms, key sizes, and compression utilities. To access the gpg and collect the statistical data, the gpgTester program, written in $C^{++,}$ served as a front-end to gpg. The gpgTester executed different cryptographic algorithms with various sized files. The gpgTester was run in a cygwin environment on a Dell Latitude D820 Microsoft Windows XP Professional with Service Pack 2. The laptop had an Intel Core 2 CPU at 2.0 GHz with 3.25 GB of RAM. Cygwin, a Linux emulator for Windows, included an executable version of gpg (Gnu Privacy Guard version 1.4.5) already installed. GpgTester used system calls to gpg to select different cryptographic algorithms. See Appendix 1 for an explanation of the gpgTester code and a brief explanation of gpg. GpgTester was able to test symmetric and public key algorithms for encrypting and decrypting files. It also tested compressed versus non-compressed files of various sizes.

3.2.2 gpgTester Scenarios.

To generate data for the MatLab optimizations, gpgTester fed different combinations of file sizes and cryptographic algorithms into gpg. The different

algorithms were Elg-E, RSA, AES, AES192, AES256, TwoFish, BlowFish, 3DES, and CAST5. Elg-E (ElGamal) and RSA are public key algorithms while the other ones are symmetric key algorithms. Data generated from the scenarios included the encryption time, decryption time, and output file size for a given input file size for the various encryption algorithms. Compression times for the algorithms were also captured for some input file sizes.

For Elg-E and RSA the key strengths were between the sizes of 1024 and 4096 in increasing increments of 256. In addition, ELG-E included key strength 768, however, RSA could not go below 1024 in gpg.

A limiting factor for symmetric key strengths was the gpg itself. Gpg allows for key strengths of 128, 192, and 256 for AES, but not for TwoFish, BlowFish, and CAST5 which are all defaulted to 128.

The file sizes chosen for encryption were 1, and 5 to 100 in increasing increments of 5, and 200 to 1000MB in increasing increments of 100. Because of the extensive amount of time in testing these various sized files, the file sizes for decryption were from 1 to 100MB for key sizes 2048 and under, and 1MB for key sizes above 2048. Primarily, the research focused on the time required to encrypt files and the encrypted output file size. In addition, if compression was used then all three of the compression algorithms were compared, i.e. bzip2, zip, and zlib. Each scenario was run 30 times using gpgTester to allow the averaged data to be used in the other steps.

Two types of files were used, randomly-generated and non-randomly generated. The randomly generated files were created with 256 bytes of data increments. The

majority of testing was based upon files generated randomly except for the testing comparisons for the compression files. Therefore to test the different compression algorithms with each other and uncompressed files, non-random sized files were created. The non-random files had various text in it including the American Constitution, the Declaration of Independence, passages from various religious texts (Christian and Buddhist), full text of Requests for Comments, and other text found on the internet. The same file (either 1, 10, 20, or 30 MB) was used to test the uncompressed file with the bzip2, zip, and zlib compression utilities. In addition, to show the offset for the compression of random files, the bzip2, zip, and zlib were compared with various sized random files.

The following table shows a depiction of the testing scenarios for one algorithm, CAST5:

Table 1: gpgTester Scenario

Algorithm	Key size	Random File size	Non-Random	Compression	Encrypted	Decrypted
CAST5	128	1	1	Bzip2, zlib, zip	Yes	Yes
		5			Yes	Yes
		10	10	Bzip2, zlib, zip	Yes	Yes
		15			Yes	Yes
		20	20	Bzip2, zlib, zip	Yes	Yes
		25			Yes	Yes
		30	30	Bzip2, zlib, zip	Yes	Yes
		35			Yes	Yes
		40		Bzip2, zlib, zip	Yes	Yes
		45			Yes	Yes
		50		Bzip2, zlib, zip	Yes	Yes
		55			Yes	Yes
		60		Bzip2, zlib, zip	Yes	Yes
		65			Yes	Yes
		70		Bzip2, zlib, zip	Yes	Yes
		75			Yes	Yes
		80		Bzip2, zlib, zip	Yes	Yes
		85			Yes	Yes
		90		Bzip2, zlib, zip	Yes	Yes
		95			Yes	Yes
		100		Bzip2, zlib, zip	Yes	Yes
		200			Yes	No
		300			Yes	No
		400			Yes	No
		500			Yes	No
		600			Yes	No
		700			Yes	No
		800			Yes	No
		900			Yes	No
		1000			Yes	No

The following data was tested via gpgTester:

- CAST5 with Key size 128. File sizes from 1, and 5 to 100 in increments of 5MB, and 200 to 1000MB in increments of 100. Decryption for file sizes 100MB and under. Compression for files 1, 10, 20, and 30 MB.
- BlowFish with Key size 128. File sizes from 1, and 5 to 100 in increments of 5MB, and 200 to 1000MB in increments of 100. Decryption for file sizes 100MB and under. Compression for files 1, 10, 20, and 30 MB..
- TwoFish with Key size 128. File sizes from 1, and 5 to 100 in increments of 5MB, and 200 to 1000MB in increments of 100. Decryption for file sizes 100MB and under. Compression for files 1, 10, 20, and 30 MB.
- 3DES with Key size 128. File sizes from 1, and 5 to 100 in increments of 5MB, and 200 to 1000MB in increments of 100. Decryption for file sizes 100MB and under. Compression for files 1, 10, 20, and 30 MB.
- AES with Key size 128, 192, and 256. File sizes from 1, and 5 to 100 in increments of 5MB, and 200 to 1000MB in increments of 100. Decryption for file sizes 100MB and under. Compression for key size 128 for files 1, 10, 20, and 30 MB.
- RSA with Key length 1024, 1280, 1536, 1792, 2048, 2304, 2560, 2816, 3072, 3328, 3584, 3840, and 4096. File sizes from 1, and 5 to 100 in increments of 5MB, and 200 to 1000MB in increments of 100. Decryption only for file sizes 100MB and under with key lengths 2048

and below and for 1MB files for the other key lengths. Compression for key length 1048 for files 1, 10, 20, and 30 MB.

- ElGamal (Elg-E) with Key length 768, 1024, 1280, 1536, 1792, 2048, 2304, 2560, 2816, 3072, 3328, 3584, 3840, and 4096. File sizes from 1, and 5 to 100 in increments of 5MB, and 200 to 1000MB in increments of 100. Decryption only for file sizes 100MB and under with key lengths 2048 and below and for 1MB files for the other key lengths. Compression for key length 1048 for files 1, 10, 20, and 30 MB.

With gpgTester, the public key algorithm is chosen with another algorithm when a system call to gpg occurs. The choices are RSA with RSA, RSA with Elg-E, DSA with RSA, or DSA with Elg-E. Regardless of which algorithm is chosen first, it is the second algorithm that is actually used for encryption. The first algorithm is strictly for signing and is not factored into the test results nor used in any timings. Therefore, only RSA with RSA and RSA with ELG-E were used and not DSA as the signing algorithm. For symmetric key, 3DES, AES, TwoFish, BlowFish, and CAST5 were used. The default key size for 3DES, TwoFish, BlowFish, and CAST5 was 128 bit (as listed in the Assumptions/Limitations section of this document).

GpgTester created random files of the requested file size. This random file was then encrypted and, if required, decrypted and/or compressed. All data was sent to a Microsoft Excel file for preliminary analysis and as input into MatLab optimizations.

A graphical representation is shown in Figure 4.

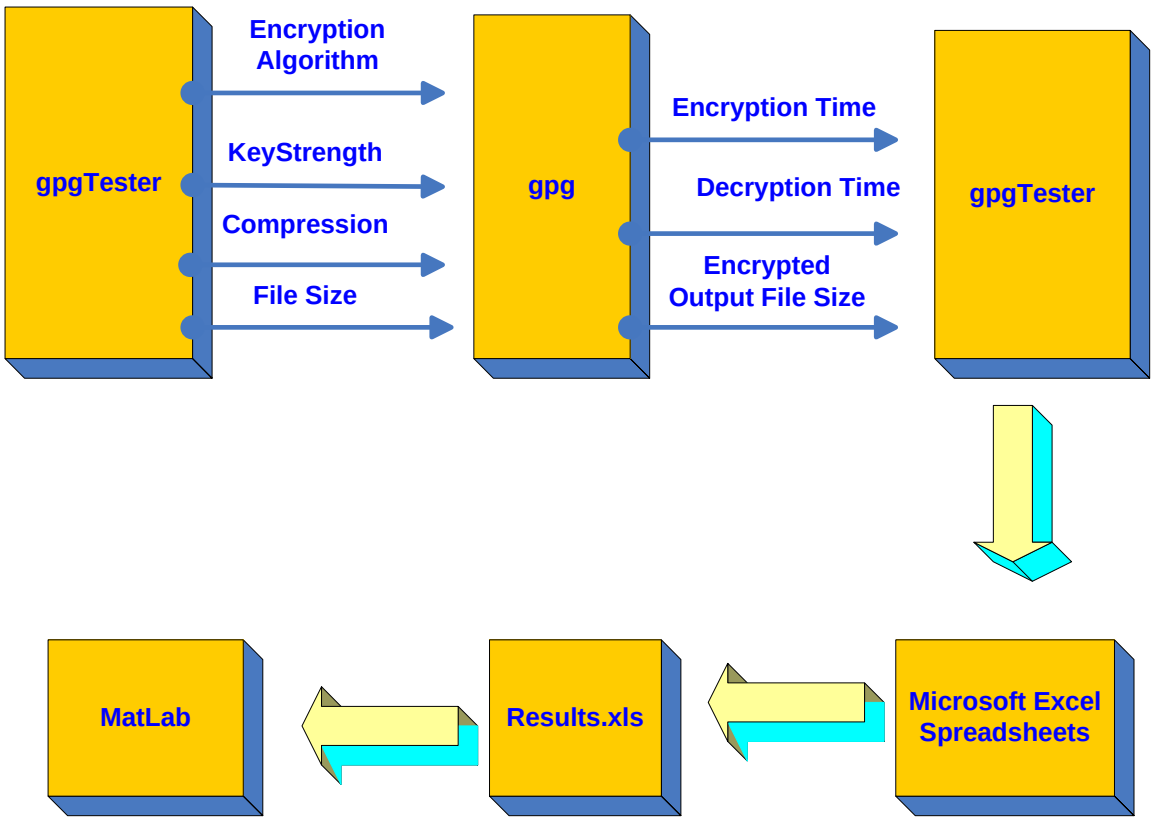

Figure 4: Initial Data Generation

3.2.3 gpgTester Verification.

Prior to using the results from the gpgTester, the program was verified to determine that the correct data would be generated.

The gpgTester created different file sizes based upon the input of the user. For a detailed explanation of the verification of the gpgTester see Appendix 1.

3.3 MatLab Interpolation (Step Two)

3.3.1 Overview of Interpolation.

The statistical data (encryption time and encrypted file size) collected from Step One was the foundation for the other steps primarily Step Two. The overall objective of

this step was to calculate the simulated encryption time and simulated encrypted file size for different commodities' file sizes. This step used interpolation to determine new data points (encryption time and encrypted file size) for different input file sizes within the range of the collected data (test results from the gpgTester, Step One). The interpolated data points for the commodities were sent to the various controllers (Step Three) for optimizing. (The rest of the research (Step Two and above) focused only on encryption times and encrypted output file size, however the data could be expanded to include decryption.) A summary graph of this step is provided in Figure 5.

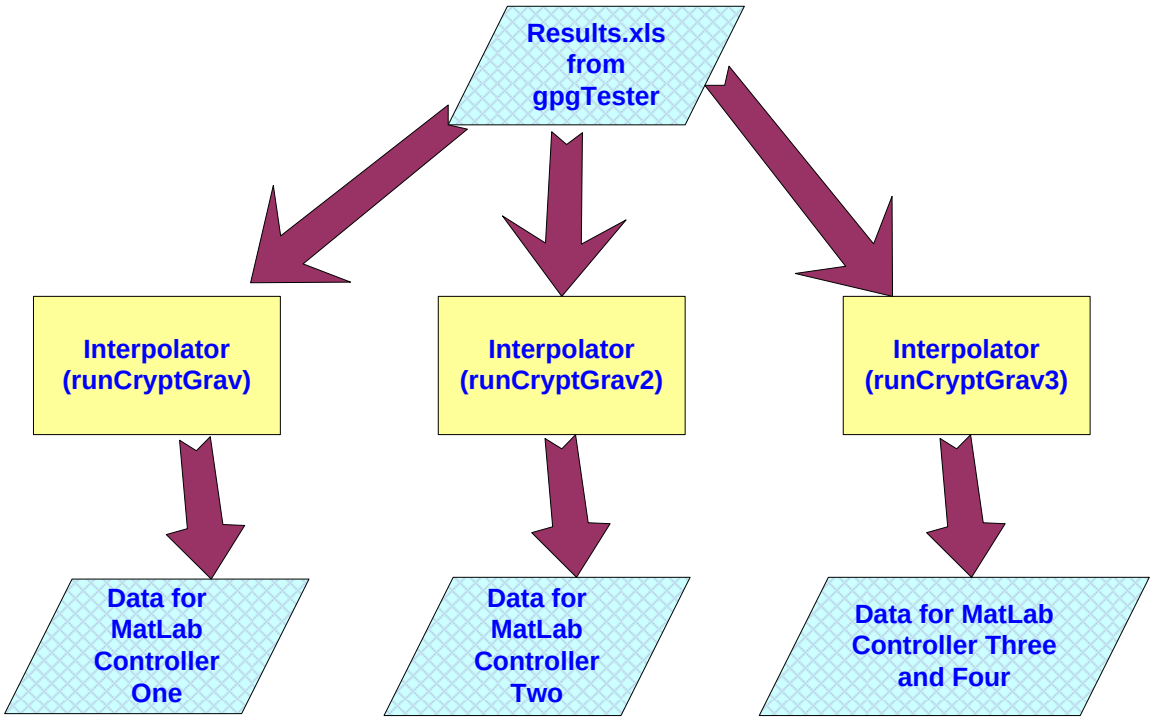

Figure 5. Data Interpolation Overview

3.3.2 Transfer Data to MatLab File.

After collecting the data from Step One, gpgTester, MatLab was used to interpolate the data. In order to actually use the data for MatLab, the data from Step One was compiled into one spreadsheet, results.xls, which included all of the averaged data (encryption times and encrypted file sizes) from the 256 files from gpgTester executions. This spreadsheet was then read by six MatLab files (inputCreate3DES, inputCreateAES, inputCreateBlowFish, inputCreateTwoFish, inputCreateCAST5, inputCreateElg, inputCreateRSA) to extract data into one file. This file was used to create one matrix in a format easily translated by other MatLab files. (An interesting characteristic of MatLab is that all data is considered matrices, therefore all of the data to and from MatLab was written in matrix (or vector) form.)

3.3.3 Data Interpolation.

To exploit the merged matrix data, an interpolation was required to determine the output for any given file size based upon the encryption algorithm, the compression function, the key size, and randomness of the file. This allows for the curve fitting of new data points within the range of the tested data from Step One. This interpolation was done within the runCryptGrav files which were the fundamental MatLab files for this part of the research. The interpolation chosen was the cubic spline interpolation using polynomials because "a practical feature of cubic splines is that they minimize the oscillations in the fit between the interpolating points" (Guenther 2002). "Polynomials are the approximating functions of choice when a smooth function is to be approximated locally" (MatLab Website). Normally, interpolation with high-order polynomials (to

create smooth lines/curves between two data points) yield erratic results, however, by using cubic splines, this erratic behavior was eliminated (Hanselman and Littlefield, 2005). Cubic spline data interpolation involved the approximation of piecewise third-order polynomials which passed through a set of existing data points. The interpolation of data points which divided the "interval into a set of subintervals and constructed a lower-degree approximating polynomial on each subinterval" (Guenther, 2002). Basically cubic splines look for cubic polynomials that approximate the curve between each pair of data points. For example, given tested input sizes of approximately 1, 5, 10, 15, 20, etc. for a random file with encryption scheme of AES 128-bit with no compression, the cubic spline from MatLab would output the encryption time and output encrypted file size for other non-tested input sizes (for example 13 or 67 MB) for that particular encryption scheme.

The mathematical calculations used by the MatLab spline involve a "unique piecewise cubic polynomial with two continuous derivatives with breaks at all interior data sites except for the leftmost and the rightmost one (the endpoints)" (MatLab) for each data point requested. The cubic spline function searched for continuity between the data points by solving for the following:

$$S(t) \in C^{n-1}[a,b]$$

where n = 3, a and b are endpoints (Spline Website).

Since the cubic spline data interpolation sought to find a data point based upon the data results from Step One, cubic splining had two constraints placed on it to prevent errors from cubic splining data points outside of the tested data ranges. The first

constraint is that if compression (bzip2, zlib, or zip) was required, then only execute cubic spline for key sizes of 1024 or 128. The next constraint was that if the request was a non-random file then cubic spline was only executed for files under 32MB with key sizes of 1024 or 128.

3.3.4 Output File Size and Encryption Time Determination.

There were three different functions (MatLab files) for determining the encrypted output file size and the encryption time, runCryptGrav, runCryptGrav2, and runCryptGrav3. All of these files include data interpolation and the encrypted output file size and encryption time based upon parameters passed into the function. The runCryptGrav functions were used by the MatLab controller for optimizing the data as seen in Figure 5 from Section 3.3.1.

3.3.4.1 runCryptGrav.

The runCryptGrav function determined which commodities (each commodity included a file size and a priority for the file size) could be sent based upon the available CPU and available bandwidth for the security algorithm requested. The input arguments for runCryptGrav were the combined matrix from the gpgTester runs, the available CPU in seconds, available bandwidth in bytes, the required security algorithm (included the key size, compression required, and randomness of file), and the matrix of commodities that were requested (which included the priority and the input file size for each commodity). The function prioritized the goodness or value (ranged from 0 to 100) of the commodity, the higher the goodness then the higher the priority of what should be sent. After interpolating the encrypted file size and encryption time via cubic splining, it

aggregated the commodities (based upon the priorities) that could be sent within the available CPU and available bandwidth. This function's output was a matrix of the maximum commodities ready for transmission in which the aggregated encrypted file size and encryption time did not exceed the available CPU and bandwidth for the specified encryption scheme.

3.3.4.2 runCryptGrav2.

Unlike runCryptGrav, the function runCryptGrav2 only had two input parameters: the matrix of collected data from gpgTester and the matrix of communication commodities which include the commodity goodness values (priorities) and file sizes. Through data interpolation using cubic splines, runCryptGrav2 outputted the encrypted file size and encryption time for all commodities sent to it. It did not determine if the aggregated commodities exceeded the available bandwidth and CPU as the function runCryptGrav did.

3.3.4.3 runCryptGrav3.

This function was used by inputCrypt3 to prepare output data for network simulations. Similar to the other runCryptGrav functions, it used cubic spline to interpolate the data sent to it. It's input and output parameters were similar to runCryptGrav2.

3.4 MatLabController (Step Three)

3.4.1 Overview of MatLab Controller.

The interpolated data found in Step Two was used as input to create the different MatLab controllers. The controllers received the list of commodities (file sizes and

priorities) and sent these commodities to the interpolator (Step Two) to determine the actual encryption times and encrypted file sizes for the commodities. In addition to the interpolated data and the list of commodities, the controllers also used encryption scheme information to determine via optimization which commodities could actually be transmitted based upon the available bandwidth and CPU. There were four different types of controllers, inputCrypt (Controller One), inputCrypt2 (Controller Two), and inputCrypt3 (Controller Three), and Controller Four. Different MatLab functions were created to assist the controller in its optimization. The main functions were for creating the commodities, determining security and performance levels, and executing a binary integer solver for optimizing. Some functions were used by more than one controller to meet the objectives of the function. Figure 6 shows a graphical representation of the major functions (MatLab files) used by the various controllers.

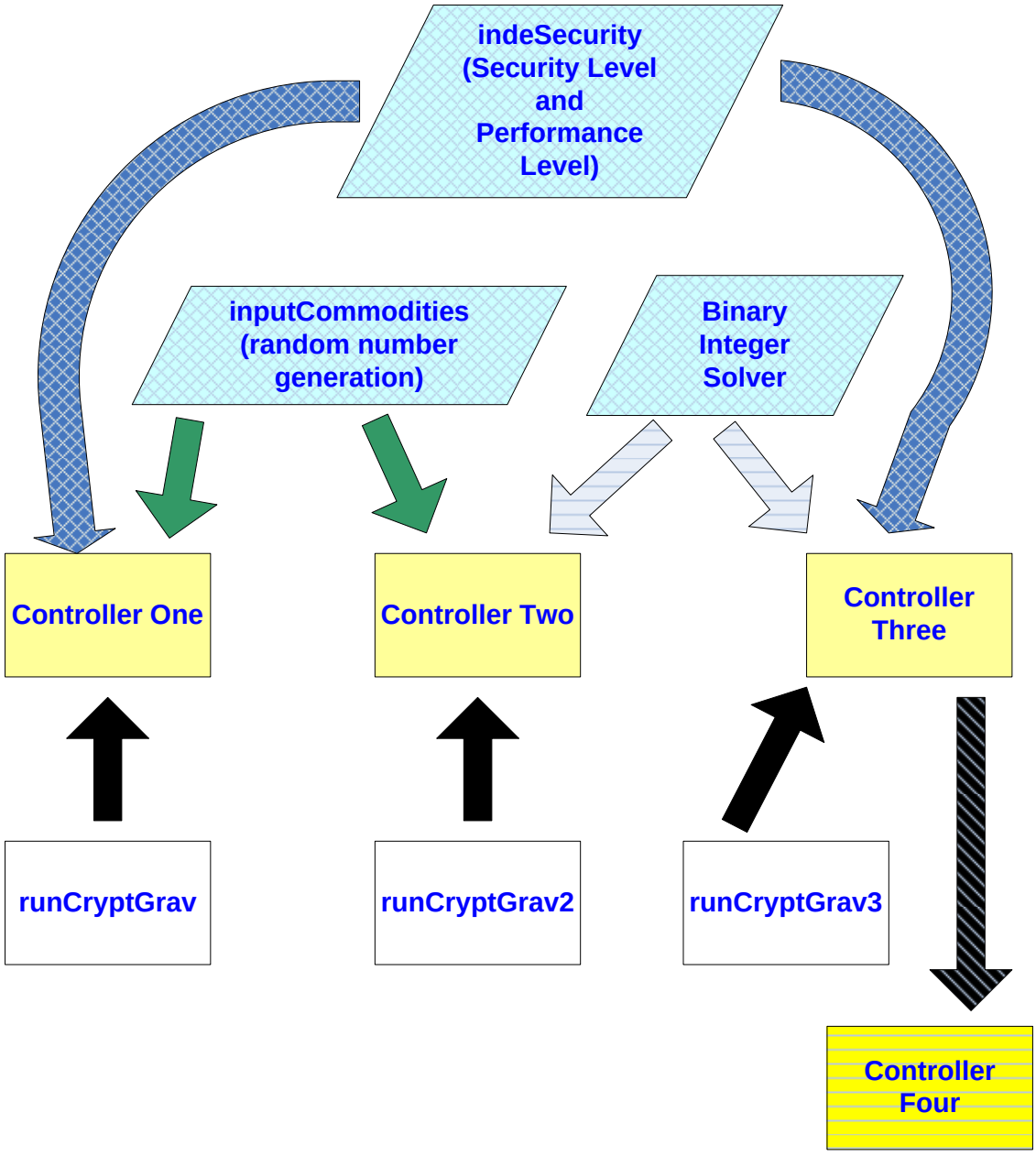

Figure 6. MatLab Controller Overview

3.4.2 Random Distribution of Commodities.

To generate commodities (each commodity had two elements in it, file size and priority) with various file sizes and priorities, an inputCommodities function was used by Controller One and Controller Two. The number of commodities requested by Controller

One and Two was sent into this function to create the matrix of commodities. A sample commodity matrix is shown below:

Table 2. Sample Commodity Matrix

File Size in Bytes	Priority
1,315,040	99
5,506,693	69
63,238,192	71
95,328,065	3

The data output from gpgTester included file sizes from 1MB to 1000MB, however, for the controllers only file sizes from 1MB to 100MB were used. This reflected more realistically with actual files used for transmissions. Rarely would file sizes of 1000MB be encrypted and sent. In addition, most files would be closer to the 1-20MB size as opposed to the 80 – 100Mb size. To randomly generate file sizes that meet the above characteristics, a gamma random number distribution was used, which generate non-negative random numbered file sizes. Randomly generated numbers via a gamma distribution "provide a fairly flexible class for modeling nonnegative random variables" (Rice, 1995) and allow for a distribution that was similar to a reverse exponential histogram.

Gamma distribution was theoretically based on the gamma function, a mathematical function defined in terms of an integral (Milton and Arnold, 2003). The general gamma distribution that MatLab used for the function, gamrnd, was

$$g(t) = \frac{\lambda^{\alpha}}{\Gamma(\alpha)} t^{\alpha-1} e^{-\lambda t}, \; t \geq 0$$

The above gamma distribution uses the gamma function, Γ, whose formula was

$$\Gamma(x) = \int_0^{\infty} u^{x-1} e^{-u} du, \; x > 0$$

To generate random numbers with a heavier distribution between 1 and 100 MB, the shape and the scale of the gamma distribution were modified. The shape, α, changed the shape of the density function while the scale, λ, changed the units of measurement. For the gamrnd MatLab function, shape was .75 and the scale was 65. If the numbers were not between 1 and 100MB, then it was not added to the list of commodities.

The value or priority for the commodity was a uniformly distributed random number (i.e. all values had an equal probability of occurring) ranging from 1 to 100. The density function for this uniform distribution was

$$f(x) = \frac{1}{n}$$

where n was a positive integer.

Each number generated was rounded up to the next integer to yield random numbers between 1 and 100.

The output commodity matrix (file size and associated priority) was returned to the calling function, either inputCrypt or inputCrypt2.

3.4.3 inputCrypt Controller (Controller One)

This function, inputCrypt, was the simplest of the controllers. It read in an input file and based upon the data in this file, outputted the randomly generated commodities that would fulfill the restrictions.

A sample input file is shown in Table :

Table 3. Sample Input File for Controller One

Available Bandwidth in MB	Available CPU in Seconds	Number of Commodities	Security Level	Performance Level
120	3000	20	4	3
100	2400	30	5	2
90	2000	40	1	1
200	1900	57	2	1
50	4000	60	3	2
300	6000	70	4	2
70	3400	83	5	3
75	2000	90	3	3
30	1900	100	2	2
20	3000	26	1	2

For each line in the file, Controller One via the inputCommodities function created the required number of commodities. Each commodity included the file

size and the priority for the commodity. The file sizes were randomly generated via a gamma distribution while the priorities were randomly generated via a uniform random distribution. Next, Controller One used the security level and performance level required (read from the input file) to determine which encryption algorithm(s) to use by calling the indeSecurity function.

The indeSecurity function took in the desired security level and performance level and outputted the encryption algorithm to use and the key size. The security level ranged from high security (5) to low security (1) whereas the performance level ranged from high (3) to low (1). Security level was 5 for High Security, 4 for Medium High, 3 for Medium, 2 for Medium Low, and 1 for Low. The performance level was 3 for fast performance, 2 for medium performance, and 1 for slow performance. Figure 7 shows the conceptual idea of security and performance dials for a dialable cryptographic controller.

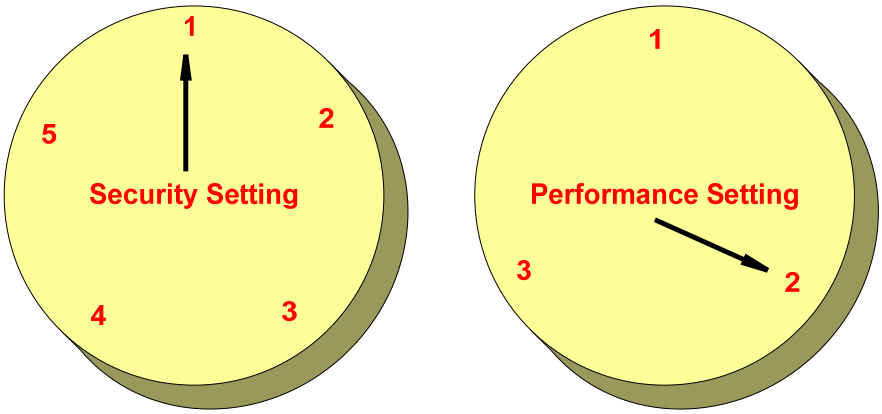

Figure 7. Dial Settings for Controller One

Once the encryption algorithm and key sizes were determined from the function indeSecurity, Controller One calls the runCryptGrav from Step Two with the input parameters. The input parameters were the commodities (input file size and priority), encryption algorithm, key size, available bandwidth, and available CPU. The runCryptGrav calculated the encryption times and output encrypted file sizes for the commodities that could be sent.

The output matrix from the runCryptGrav served as a decision tool for the Controller One (inputCrypt). Normally there was more than one output matrix because the security and performance settings had more than one encryption scheme associated with it. The inputCrypt controlled which scheme was the best one to use. It first maximized the number of commodities to be sent, then chose the smallest combined encryption time for the commodities, and finally the greatest output file size. For example if there were three different encryption algorithms available, but one encryption line could send nine commodities but another could only send 5, then the controller would chose the encryption algorithm that yielded nine commodities. However, if two different encryption algorithms yield the same number of commodities, then the controller would chose the one with the smallest combined encryption time.

Finally, inputCrypt wrote the output commodities to a file for the user. The output file repeated the available bandwidth, available CPU, number of commodities, security level, and performance level used. In addition, it wrote out the requested commodities (input file size and priority), the list of commodities that could be sent (input file size, priority, output encrypted file size, and encrypted time), encryption

algorithm and key size used, the total goodness (sum of the chosen commodities' priorities) of the commodities, the total bandwidth required, and the total encryption time needed.

3.4.4 inputCrypt2 Controller (Controller Two)

Controller Two introduced binary integer programming to determine the best encryption algorithms to use. "Binary integer programming problems involve minimizing a linear objective function subject to linear equality and inequality constraints. Each variable in the optimal solution must be either a 0 or a 1. The MatLab Optimization Toolbox solved these problems using a branch-and-bound algorithm that searches for a feasible binary integer solution, updates the best binary point found so far as the search tree grows, and verifies that no better solution is possible by solving a series of linear programming relaxation problems" (MatLab Website). The controller did not use the concept of security and performance levels but rather presented seven encryption algorithms per commodity for the binary integer solver to optimize.

In this section, the idea of binary choice (either 0 or 1) referred to whether or not the commodity could be sent (0 if not sent and 1 if sent). The decision was represented by a binary variable *x*. The object was to maximize $f'x$ such that $Ax \leq b$.

Each commodity was assigned to an x such that $x_j = \begin{cases} 1 \\ 0 \end{cases}$. The priorities (also known as values) of the commodities were used as weights for the binary integer programming as the objective coefficients function, *f*. The total net value of the decisions were represented by $f'x$ or Z.

Because MatLab works with only matrix manipulations, all data was changed into the appropriate matrix form.

Different commodities were tested using seven encryption algorithms with key size of 1024 for public key and 128 for symmetric key (e.g. if there were 30 commodities, then the solution would try to maximize with 210 (30 x 7) commodities).

The general commodity matrix for n commodities would be:

$$c = \begin{pmatrix} i_1 & p_1 & s_{11} & t_{11} & s_{12} & t_{12} & \cdots & s_{17} & t_{17} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ i_n & p_n & s_{n1} & t_{n1} & s_{n2} & t_{n2} & \cdots & s_{n7} & t_{n7} \end{pmatrix}$$

The first column, (i), was the input file size, the second column, (p), was the priority of the commodity, the rest of the columns represented the output encrypted file size (s) and encryption time (t) for each of the seven algorithms tested.

The general matrix equation for f (the vector containing the objective function coefficients) was

$$f = \begin{pmatrix} p_1 \\ \vdots \\ p_{n*7} \end{pmatrix}$$

where the vector, f, corresponded to the priorities in the commodity, c, matrix . Notice that f had seven times more variables than the number of commodities to account for the seven different encryption algorithms per commodity.

The general b vector (right hand side values of $Ax \leq b$) was

$$b = \begin{pmatrix} 1_1 \\ \vdots \\ \vdots \\ 1_{n*7} \\ BW \\ CPU \end{pmatrix}$$

where BW represents available bandwidth in bytes and CPU represents available CPU in seconds. The 1's represent the at-most value for each commodity (only at most one of the seven encryption algorithms can be chosen for each commodity).

The general *A* matrix, constraint coefficient matrix, was:

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & \cdots & 0_{n*7} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & \cdots & 0_{n*7} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0_{n*7} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0_n & 0_n & 0_n & 0_n & 0_n & 0_n & 0_n & 0_n & 0_n & \cdots & 1_{n*7} \\ size_1 & & \cdots & & & & & & & & size_{n*7} \\ time_1 & & \cdots & & & & & & & & time_{n*7} \end{pmatrix}$$

The function chose at most one encryption algorithm out of seven available for each commodity, therefore each commodity row had seven consecutive ones in it. The last two rows were the encrypted file size and encryption time for each commodity's encryption algorithm.

The general solution for n commodities was $Z = -f_1x_1 - f_2x_2 - \cdots\cdots - f_{n*7}x_{n*7}$ which sought to maximize the function, $Ax \le b$, to find the *x* variables.

The general *x* vector solution was:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{n*7} \end{pmatrix}$$

where each x_n was either a 0 or 1 with the imposed constraint that each commodity row from *A* had at most one *x=1*.

The above explanations were for commodities with seven available algorithms to choose from. For example (to show how Controller Two worked), given four commodities (each commodity includes file size and priority for commodity) with two encryption schemes available for each commodity, an available bandwidth of 119, and an available encryption time of 79, (not actual data numbers used for this example), the commodities would be sent to the interpolator to determine encrypted file size and encryption time for the two encryption schemes for each commodity. The interpolated data would be included in the commodities matrix, *c*. In addition to this matrix, the other matrices would be created prior to calling the binary integer programming function:

$$c = \begin{pmatrix} 10 & 22 & 15 & 31 & 17 & 38 \\ 20 & 32 & 25 & 41 & 27 & 48 \\ 30 & 42 & 35 & 51 & 37 & 58 \\ 40 & 52 & 45 & 61 & 47 & 68 \end{pmatrix} \qquad f = \begin{pmatrix} 22 \\ 22 \\ 32 \\ 32 \\ 42 \\ 42 \\ 52 \\ 52 \end{pmatrix} \qquad b = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 119 \\ 79 \end{pmatrix} \qquad x = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 15 & 17 & 25 & 27 & 35 & 37 & 45 & 47 \\ 31 & 38 & 41 & 48 & 51 & 58 & 61 & 68 \end{pmatrix}$$

A constraint placed on the binary function was that only one of the encryption algorithms could be used per commodity, i.e. mutually exclusive with at most one encryption algorithm chosen. Another constraint was that the sum of all the commodities' output encrypted file sizes had to be less than the available bandwidth. Furthermore, the sum of all of the commodities' encrypted times had to be less than the available CPU. These three constraints were reflected in the above sample matrices.

When the function was called, it attempted to maximize the number of *x*'s (1s) for the four commodities. A detailed explanation of the binary integer programming follows based upon the above matrices.

$$c = \begin{pmatrix} 10 & 22 & 15 & 31 & 17 & 38 \\ 20 & 32 & 25 & 41 & 27 & 48 \\ 30 & 42 & 35 & 51 & 37 & 58 \\ 40 & 52 & 45 & 61 & 47 & 68 \end{pmatrix}$$

The above matrix is the commodity matrix with the first column representing the input file size, 2nd column represents the value placed on each commodity, the 3rd column is the output encrypted file size for encryption algorithm A, the 4th column is the encryption time for algorithm B, the 5th column is the output encrypted file size for algorithm B, and the last column is the encryption time for algorithm B.

$$f = \begin{pmatrix} 22 \\ 22 \\ 32 \\ 32 \\ 42 \\ 42 \\ 52 \\ 52 \end{pmatrix}$$

The above vector is the vector containing the objective function's coefficients. It is weighted with the values from c, the commodity matrix. Because each commodity has two available encryption algorithms, there are eight variables instead of four variables (similar to having eight separate commodities: four commodities with two different encryptions). Each value is used twice to represent the duplicated commodities.

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 15 & 17 & 25 & 27 & 35 & 37 & 45 & 47 \\ 31 & 38 & 41 & 48 & 51 & 58 & 61 & 68 \end{pmatrix}$$

The above matrix, A, is the primary matrix. It contains the constraints' coefficients. It sets the stage for mutual exclusions for the commodities. Each commodity is a row in the first four rows of the matrix. The function will determine which encryption algorithm to choose for the commodity. It will choose at most one encryption algorithm per commodity (for mutual exclusion there are two 1's per commodity row to reflect the two available encryption algorithms). The last two rows represent the commodity's file size and encryption time respectively.

$$b = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 119 \\ 79 \end{pmatrix}$$

This vector sets the limits for the function, $Ax \leq b$. This vector, b, holds the right-hand side values for each constraint in the constraint matrix, A. The first eight rows (ones) represent the eight different commodities. The second to last row is the available bandwidth and the last row is the available CPU.

When the binary integer programming function is called it seeks to maximize the following equation for eight commodities and two encryption algorithms: $Z = -f_1x_1 - f_2x_2 - f_3x_3 - f_4x_4 - f_5x_5 - f_6x_6 - f_7x_7 - f_8x_8$ or more specifically for this example: $Z = -22x_1 - 22x_2 - 32x_3 - 32x_4 - 42x_5 - 42x_6 - 52x_7 - 52x_8$ via a branch and bound method. The output binary solution, x, is:

$$x = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

Each line in the vector, x, represents whether or not the commodity can be sent (1 for sending and 0 for not sending). Notice that for each commodity (originally four but working set is eight) only one encryption algorithm is chosen per commodity. Commodity 1 is the first and second rows of x, commodity 2 is the 3rd and 4th rows, commodity 3 is the 5th and 6th rows, and commodity 4 is the 7th and 8th rows. A line by line analysis of x shows that only commodity 1, 2, and 4 can be sent using encryption algorithm A for commodities 1 and 4 and encryption algorithm B for commodity 2.

Unfortunately, the binary integer function included with the MatLab Optimization Toolbox, bintprog, produced erratic results for large commodities. Therefore, an add-on to the MatLab software was used to perform binary integer programming. This add-on was glpk with the glpk mex interface for MatLab. The same principles for bintprog applied to glpk, however, glpk was stable for large commodities (Yalmip Website).

Similar to Controller One, this Controller Two also generated random commodities for the input file size and priority. See Section 3.3.3 for an explanation of this random generation.

Controller Two called the runCryptGrav2 with just two parameters, the commodities and a matrix of the compiled data from gpgTester. Prior to sending the parameters to runCryptGrav2, seven security lines were added to each line of the commodity. This security line included the encryption algorithm, key size, and non-compression requirement. The security lines were the following:

- 3DES 128 bit
- AES 128 bit
- BlowFish 128 bit
- CAST5 128 bit
- TwoFish 128 bit
- ElGamal 1024 bit
- RSA 1024 bit

It was not necessary to provide the available bandwidth nor available CPU to runCryptGrav2 because the binary integer function coded into Controller Two searched for the optimal solution using the available CPU and available bandwidth as part of the *b* vector.

The output from runCryptGrav2 was a matrix of the commodities sent to it, output encrypted file size, and encryption time.

From this matrix, the glpk (binary integer solver), optimized the data. The solver was restricted to only using one encryption line per commodity. Also, the solver searched for solutions that would not exceed the available bandwidth and available CPU.

The Controller Two read data from a file for the input. The data included the available bandwidth, available CPU, and the number of commodities. A sample input file is shown in Table 4.

Table 4. Sample Input File for Controller Two

Available Bandwidth in MB	Available CPU In Seconds	Number of Commodities
120	3000	4
100	2400	8
90	2000	12
200	1900	5
50	4000	20
300	6000	10
70	3400	3
75	2000	25
30	1900	9
20	3000	6
120	3000	20
100	2400	30
90	2000	40

The output from Controller Two was sent to an output file. The output was similar to Controller One's output. The output included the requested commodities (input file size and priority), the list of commodities that could be sent (input file size,

priority, output encrypted file size, and encryption time), encryption algorithm and key size used, the total goodness (sum of the chosen commodities' priorities) of the commodities, the total bandwidth required, and the total encryption time required to send the commodities.

3.4.5 inputCrypt3 Controller (Controller Three).

Controller Three was used as a basis for NS-2 simulations. The input files to the controller and the output files from this controller were in the same format as required for NS-2. It merged the binary integer programming from Controller Two with the security and performance levels of Controller One.

Unlike the other two controllers, it read in a file to determine the commodities (instead of randomly generating the commodities) that should be sent based upon the parameters in the file. The input parameters in the file included the available bandwidth and available CPU. In addition, each commodity included the commodity number, input file size, the priority of the commodity, the security level, and performance level. A sample input file is shown in Table 5. The first row of the table shows the available bandwidth and the available CPU. The next three columns (0) of the first row are placeholders to ensure matrices are formed correctly for MatLab. The next rows show the parameters for each commodity: commodity number, input file size, the priority of the commodity, the security level, and performance level.

Table 5. Sample Input File for Controller Three

100	2400	0	0	0
1	10	60	3	1
2	15	28	2	2
3	30	90	3	3
4	5	30	2	1
5	50	15	4	2
6	60	75	1	1
7	3	90	5	3
8	22	49	2	2
9	20	74	2	3
10	44	38	5	2

Each security level and performance level associated with three possible encryption schemes, therefore each commodity was tripled to account for three encryption schemes. For example, the last row of Table 5 is for commodity 10 with file size 10 and priority 38. The security level, 5, and the performance level, 2, converted to ElGamal with key size 3584, RSA with key size 3840, and RSA with key size 3584.

The commodities were sent to the runCryptGrav3 with the encryption schemes. The output from runCryptGrav3 (the encrypted file size and encryption time for each

commodity based upon the encryption scheme) was used as the input for the binary integer function.

Instead of seven different encryption schemes as in Controller Two, there were only three encryption schemes. However, for Controller Three, the range of encryption algorithms with key size (768 – 4096 for public key and 128 – 256 for symmetric key) was greater than with Controller Two. (Controller Two only used key sizes of 128 or 1024 for the seven encryption algorithms.)

As part of the binary integer programming (see Section 3.4.4 for a detailed explanation of how the research utilized binary integer programming) the commodity matrix, c, for n commodities was (all commodities were tripled to account for three encryption algorithms per commodity):

$$c = \begin{pmatrix} i_1 & p_1 & s_{11} & t_{11} & s_{12} & t_{12} & s_{13} & t_{13} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ i_n & p_n & s_{n1} & t_{n1} & s_{n2} & t_{n2} & s_{n3} & t_{n3} \end{pmatrix}$$

The general coefficient matrix, *f*, for n commodities was

$$f = \begin{pmatrix} p_1 \\ \vdots \\ p_{n*3} \end{pmatrix}$$

The coefficient matrix was used to solve $Z = -f_1x_1 - f_2x_2 - \cdots\cdots - f_{n*3}x_{n*3}$ equation to maximize the *x's*. The solver was confined by the constraint's equation, $Ax \le b$, where *A, x,* and *b* respectively were:

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0_{n*3} \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & \cdots & 0_{n*3} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \cdots & 0_{n*3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0_n & 0_n & 0_n & 0_n & 0_n & 0_n & 0_n & 0_n & 0_n & \cdots & 1_{n*3} \\ size_1 & & \cdots & & & & & & & & size_{n*3} \\ time_1 & & \cdots & & & & & & & & time_{n*3} \end{pmatrix}$$

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{n*3} \end{pmatrix}$$

$$b = \begin{pmatrix} 1_1 \\ \vdots \\ \vdots \\ 1_{n*3} \\ BW \\ CPU \end{pmatrix}$$

The vector, *b*, includes the available bandwidth and CPU from the inputted file. The vector, *x*, corresponds to the commodities that can be sent. If *x=1*, then the commodity can be sent, however, if *x=0*, then the commodity will not be sent. In addition, the matrix, *A*, enforced mutual exclusion (at most one) for the three different encryption schemes.

The output from this controller was sent to a file in the same format as would be sent to the NS-2 for simulations. The output was a matrix of the commodities (including original commodity numbers) that could be sent based upon the available bandwidth and CPU.

3.5 NS-2 Simulations (Step Four)

The final step after creating the controllers were to simulate commodities received from a network node. The network simulator, NS-2, simulated a small wired network of up to 10 nodes. Although the thesis is focused on wireless networks, the data gathered from the network simulator applied to a wireless network. Because NS-2 runs in a non-Windows environment while MatLab runs in a Windows environment, an executable MatLab file was required to transfer information between NS-2 and MatLab. For Step Four, the executions were completed in a Linux environment.

The controller created for Step Four was the exact same controller as Controller Three with some minor modifications. This new controller was called encryptFitter. One minor modification was an adjustment from file size to packet size. Other modifications were to align the controller's code with the format required by NS-2.

The NS-2 simulations were one extension of a research topic by John Pecarina (Pecarina, 2008). Pecarina's research focused on creating an agent based framework to maximize information available at network nodes. The research included a Hybrid Agent for Network Control (HANC) network simulator which controlled the routing decisions for the nodes by polling the network for information. The MatLab Controller Four was

integrated into HANC to expand the information available at each node to include commodities, security level, performance level, and available bandwidth and CPU.

With NS-2, the network nodes were concerned with sending packets and not actual files. Each node knew how much bandwidth was available on each available network path and the destination for the packets. A simple network diagram is shown in Figure 8 with the associated bandwidth available for each network path.

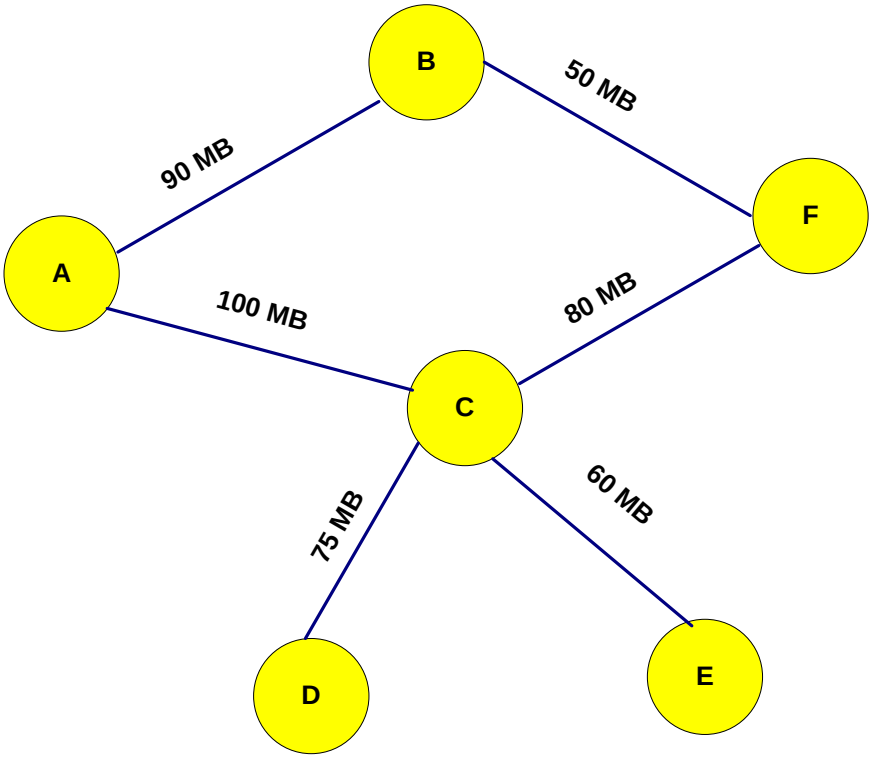

Figure 8. Simple Network Diagram

Because of this specification, all input file sizes were considered in Bytes and not MegaBytes. All data gathered from Step One, gpgTester, was from file sizes of greater than 1MB. To actually interpolate and optimize the data for packet sizes, the data was multiplied by 1MB. After interpolation and optimization, the encryptFitter divided by

1MB, to return the data back to packet sizes. This assumed that multiplying the file by 1MB would yield the correct interpolated/optimized data (e.g. 100 Bytes would yield the same results as 100Bytes multiplied by 1MB). Although this was not necessarily a reasonable assumption, it did allow for direct integration into HANC.

NS-2 via HANC wrote data to a file (encryptin.m) which was read by the MatLab controller for optimizing the data. Once MatLab completed optimizing the data via interpolation and binary integer programming, it wrote data to a different file (encryptout.m) for NS-2's HANC to read. This circular transfer was completed repeatedly for the simulations. Figure 8 is a high level view of this transfer.

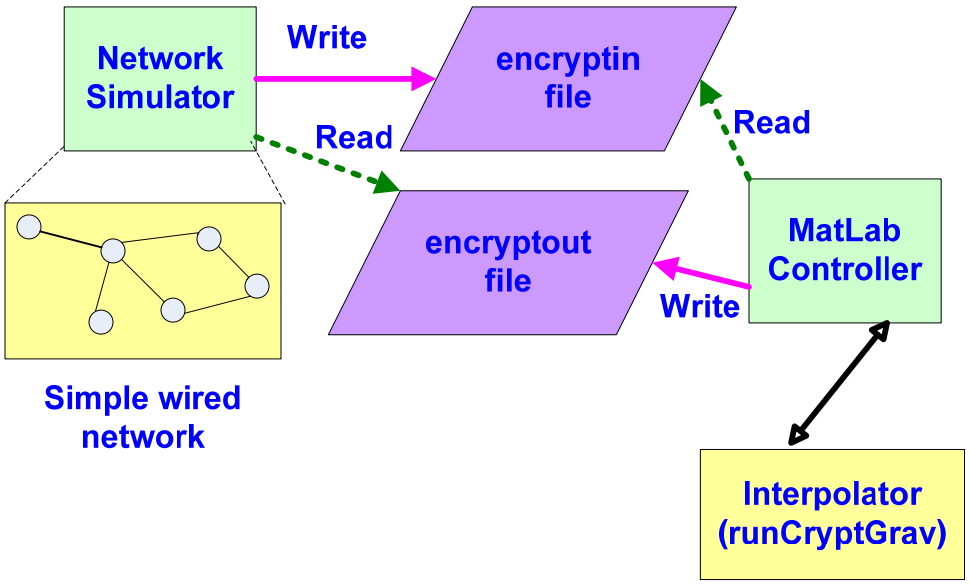

Figure 9. NS-2 and MatLab Transfer of Information

The encryptin.m file held the parameters for the MatLab controller. When the file was ready to be read, MatLab, encryptFitter (Controller Four), read it and then called the runCryptGrav4 program to interpolate the data. Finally, encryptFitter executed a binary programming solver to optimize the data. The results were written into the encryptout.m

file which was then read by NS-2 when available. The results included commodities numbers (flow numbers) that could be sent, the encrypted file sizes, and encryption times.

IV. Analysis and Results

4.1 Analysis Overview

The research was divided into four separate parts, the gpgTester executions (Step One), data interpolation (Step Two), data optimizing (Step Three), and NS-2 simulations (Step Four). Each section yielded results for analysis and also for input into the subsequent steps.

4.2 gpgTester Results (Step One)

The gpgTester program, written in C^{++} as a front end into gpg, ran different encryption scheme scenarios thirty times each. The different scenarios varied in cryptographic algorithm, file size, key strength, and compression. The symmetric key algorithms were 3DES, AES, BlowFish, CAST5, and TwoFish. The key size used was 128 bit for all of them. In addition, AES was tested with 192 and 256 bit encryption because gpg allowed 3 different key sizes (128, 192, and 256) for AES only. The public key algorithms were RSA and ELG-E with different key sizes from 1024 and 4096, with Elg-E having an additional 768 key size.

The results from gpgTester were fed into Microsoft Excel files. There were 256 separate Excel files generated via the testing. Each encryption scenario was run 30 times. A sample output of the 30 runs is listed in Table 6 for a 1MB ELG-E with 1792 key size.

Table 6. Elg-E Sample Output from gpgTester

Pub type	Pub size	Sub type	Sub size
RSA	1792	ELG-E	1792

Run	Time	Input Size	Output Size	Decryption Time	Decryption Size
1	150	1048576	1049123	170	1048576
2	140	1048576	1049123	170	1048576
3	140	1048576	1049123	170	1048576
4	150	1048576	1049123	150	1048576
5	150	1048576	1049123	170	1048576
6	140	1048576	1049123	160	1048576
7	140	1048576	1049123	160	1048576
8	140	1048576	1049123	170	1048576
9	150	1048576	1049123	140	1048576
10	140	1048576	1049123	160	1048576
11	140	1048576	1049123	170	1048576
12	140	1048576	1049123	170	1048576
13	140	1048576	1049123	160	1048576
14	140	1048576	1049123	160	1048576
15	140	1048576	1049123	170	1048576
16	140	1048576	1049123	160	1048576
17	140	1048576	1049123	160	1048576
18	140	1048576	1049123	180	1048576
19	150	1048576	1049123	160	1048576
20	140	1048576	1049123	170	1048576
21	160	1048576	1049123	170	1048576
22	140	1048576	1049123	160	1048576
23	140	1048576	1049123	160	1048576
24	120	1048576	1049123	170	1048576
25	150	1048576	1049123	160	1048576
26	140	1048576	1049123	150	1048576
27	140	1048576	1049123	170	1048576
28	140	1048576	1049123	170	1048576
29	120	1048576	1049123	160	1048576
30	140	1048576	1049123	160	1048576

Average Encryption Time	141.333
Average Decryption Time	163.667

Each scenario within gpgTester had a similar output. For a listing of all of the scenarios, see Section 3.2.2 and Table 1 from Section 3.2.2.

Some of the 256 separate Excel files held more than 15 different scenarios. These files were combined to form 16 averaged files. The total number of encryption schemes was over 1350 (with each scheme tested 30 times). A sample spreadsheet showing the average encryption time, decryption, and size difference (encrypted file size minus the original input file size) is listed in Table 7 for non-compressed and compressed scenarios (each line was considered a separate scenario) using AES.

Table 7. Sample Averaged Output for AES Scenarios

	Keysize	Input Size	Output Size	Size Difference	Average Encryption Time	Average Decryption Time
	128	1048576	1048673	97	96	128
	128	5242880	5242977	97	259	445
	128	10223616	10223713	97	449	820.367
	128	15204352	15204449	97	631	1204
	128	20185088	20185185	97	822.2	1579.67
	128	25165824	25165921	97	1026	1977.33
	128	30146560	30146657	97	1218	2367.07
	128	35127296	35127393	97	1428	2737.83
	128	40108032	40108129	97	1613.9	3118.6
	128	45088768	45088865	97	1809.3	3499.7
	128	50069504	50069601	97	2009	3877.83
	128	55050240	55050337	97	2192	4267.7
	128	60030976	60031073	97	2389.47	4623.07
	128	65011712	65011809	97	2584.43	5030.4
	128	70254592	70254689	97	2783.03	5425.17
	128	75235328	75235425	97	2965.33	5809.43
	128	80216064	80216161	97	3145.43	6192.3
	128	85196800	85196897	97	3363.07	6572.73
	128	90177536	90177633	97	3550.33	6938.87
	128	95158272	95158369	97	3742.73	7327.03
	128	100139008	100139105	97	3923.53	7716.07
bzip2						
	Keysize	Input Size	Average Output Size	Size Difference	Average Encryption Time	Average Decryption Time
	128	1162313	188701	-973612	401.667	166.333
	128	10437672	2470818	-7966854	2251	1168.33
	128	20576925	4740467	-15836458	4404	2215.67
	128	31014597	7206106	-23808491	6624	3342.93

The analysis of the data from this section included the file comparisons (input file size compared to the output encrypted file size), the encryption time, and the decryption time. In general, the stronger key size (longer key) took more time to encrypt and decrypt.

4.2.1 File Size Comparisons.

Two types of files were used, randomly-generated and non-randomly generated files (see Section 3.2.2 for an explanation of how the files were generated). Each file size was tested 30 times. The average file size difference (the encrypted file size minus the original input file size) was calculated for comparisons.

4.2.1.1 Non-Compressed File Size Differences

For the 30 different runs, the file size difference remained the same when non-compression was chosen (very infrequently, a size difference of 1 could be noticed) regardless of the initial size of the file. In addition, the non-randomly generated file difference was less than the randomly generated file size. For the symmetric key algorithms, CAST5, BlowFish, and 3DES had the same file difference (65 Bytes for random files and 54 Bytes for non-random files), while TwoFish and AES yielded the same result of 97 Bytes for random files or 86 Bytes for non-random files. Interestingly enough, the different key sizes for AES (128, 192, and 256) did not change the file size difference. A sample output for symmetric keys AES, BlowFish, CAST 5, and TwoFish is shown in Table 8.

Table 8. Sample File Size Differences for Symmetric Algorithms

Encryption Algorithm	Key Size	*Random*	Input File Size in Bytes	Output File Size in Bytes	File Size Difference in Bytes
AES	128	*Yes*	1048576	1048673	97
	128	*Yes*	5242880	5242977	97
	192	*Yes*	55050240	55050337	97
	192	*Yes*	60030976	60031073	97
	256	*Yes*	400031744	400031841	97
	256	*Yes*	500170752	500170849	97
TwoFish	128	*Yes*	1048576	1048673	97
	128	*Yes*	10223616	10223713	97
BlowFish	128	*Yes*	80216064	80216129	65
	128	*Yes*	85196800	85196865	65
CAST5	128	*Yes*	70254592	70254657	65
	128	*Yes*	80216064	80216129	65
AES	128	*No*	1162313	1162399	86
	128	*No*	10437672	10437758	86
TwoFish	128	*No*	20576925	20577011	86
	128	*No*	31014597	31014683	86
BlowFish	128	*No*	20576925	20576979	54
	128	*No*	31014597	31014651	54
CAST5	128	*No*	1162313	1162367	54
	128	*No*	10437672	10437726	54

When compared to the public key algorithms, all of the symmetric key algorithms had a significantly smaller file size difference. Figure 10 shows a comparison of the file size differences for the symmetric and public key algorithms.

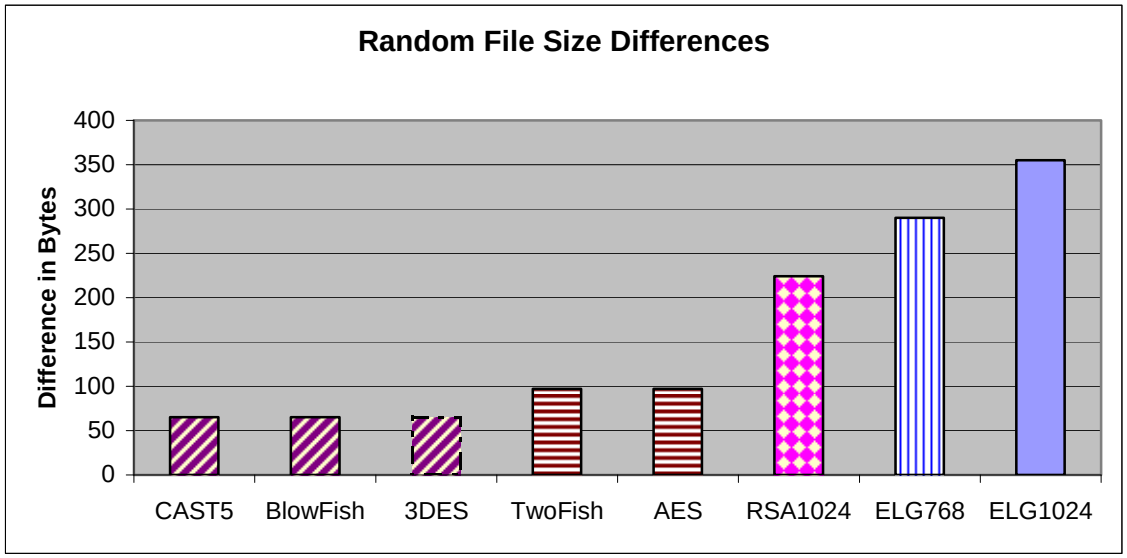

Figure 10. File Size Differences for Random Files

Unlike AES, the increase in key size increased the file size differences for RSA and ELG-E. For example, Elg-E with key size 768 had a smaller file size difference than Elg-E with 1024 key size. Of the two public key algorithms, RSA had smaller file size differences than ElGamal for all key sizes between 1024 and 4096 (RSA was not tested at key size 768). Figure 11 shows the file size differences for RSA and ElGamal for the different key sizes (all file size differences were greater than the symmetric key differences).

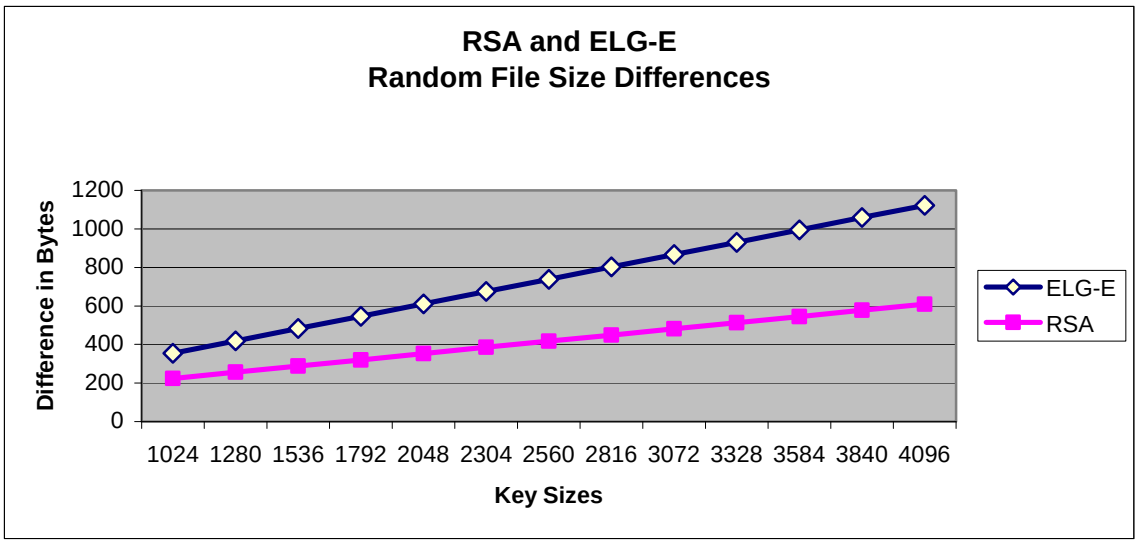

Figure 11. Comparison of Public Key File Size Differences

Similar to the symmetric key algorithms (AES, 3DES, CAST5, BlowFish, and TwoFish,), the file size differences were slightly less for non-randomly generated files. For example, RSA with 1024 key size was 213 Bytes (file size difference) for a non-random file size as opposed to 224 Bytes for a randomly-generated file.

4.2.1.2 Compressed File Size Differences

The compression of the input file sizes greatly impacted the resulting file size difference especially for randomly generated files.

The compression of randomly generated files actually yielded a larger file than a non-compressed file. Two reasons for this are that the compression algorithms (bzip2, zip, and zlib) were based upon patterns within the files (no patterns mean less compression) and that the compression utility itself adds overhead to the outputted encrypted file. The randomly generated files showed little repetitive patterns in the file size differences except for the zlib test scenarios (for non-compressed, the same file size

difference could be seen regardless of initial file size). A sample comparison of non-compressed versus compressed file size differences is shown in Table 9 for randomly-generated files using TwoFish.

Table 9. Sample Random File Size Difference Comparison

TwoFish	Input File Size in Bytes	Output File Size in Bytes	File Size Difference in Bytes
Non-Compressed	1048576	1048673	***97***
	10223616	10223713	***97***
	20185088	20185185	***97***
	30146560	30146657	***97***
	40108032	40108129	***97***
	50069504	50069601	***97***
bzip2	1048576	1053868	***5292***
	10223616	10273894	***50278***
	20185088	20283678	***98590***
	30146560	30293862	***147302***
	40108032	40303630	***195598***
	50069504	50313807	***244303***

The above table shows the file size difference as constant for non-compressed (discussed in the previous section), however, with the compression algorithms, there were no constant file size differences. As the input file size increased, the file size difference increased (significantly) as well. On the other hand, for non-randomly generated files, the file size difference for compression actually decreased because the compression utility was able to actually compress the file by finding patterns within the text of the file.

Table 10 shows a sample output file size difference for non-randomly generated files for TwoFish using non-compressed and bzip2 compression.

Table 10. Sample Non-Random File Size Difference Comparison

TwoFish	Input File Size in Bytes	Output File Size in Bytes	File Size Difference in Bytes
Non-Compressed	1162313	1162399	86
	10437672	10437758	86
	20576925	20577011	86
	31014597	31014683	86
bzip2	1162313	188688	-973625
	10437672	2470819	-7966853
	20576925	4740311	-15836614
	31014597	7206048	-23808549

The random graph for the file size differences (sample graph for RSA is shown in Figure 12) show that bzip2 yielded the greatest file size difference (biggest output encrypted file size) while zlib was the smallest (i.e., zlib was better at compressing randomly generated files).

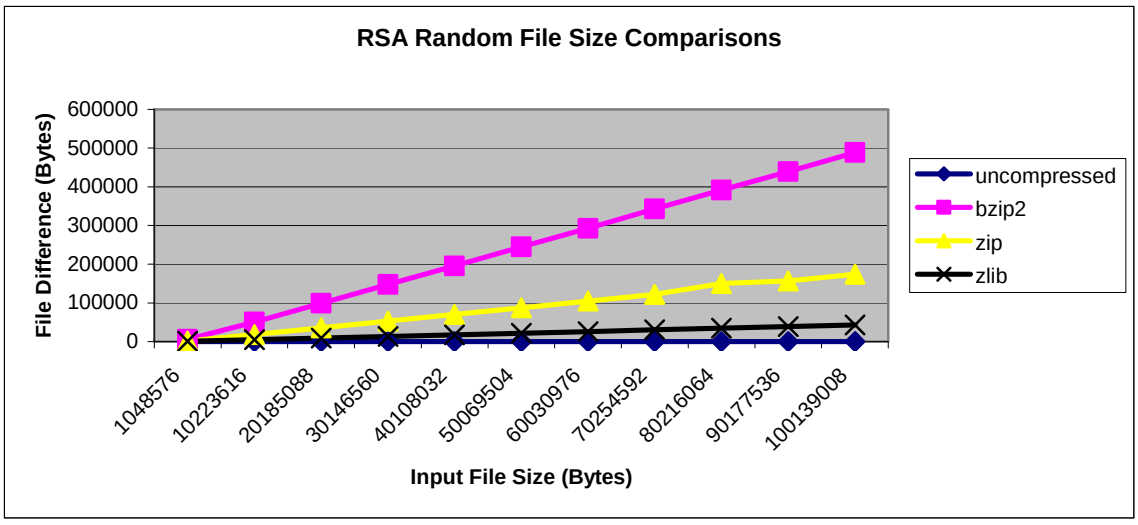

Figure 12. RSA Random File Size Compression Comparisons

The other cryptographic algorithms would display graphs similar to the above graph. The uncompressed output file sizes were far below the other compressed files and the uncompressed file size difference remained the same for the algorithm despite input file size increases, unlike the compressed algorithms. Zlib exhibited the smallest output file size difference for compression algorithms followed by zip.

For the non-random file sizes, bzip2 outperformed the other compression utilities (zip and zlib) for compressing the files (i.e. a higher negative number corresponded to a smaller compressed file). Zip was better than zlib for compressing random files. The files compressed created a significantly smaller file as can be seen by the negative outcomes in Figure 13. The graph was representative of the other algorithms because there was no significant difference between the algorithms.

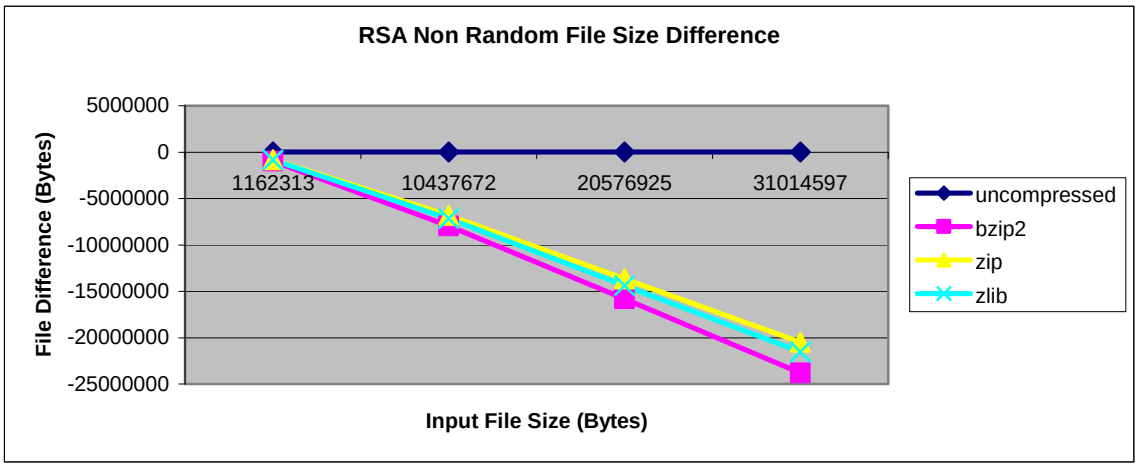

Figure 13. RSA Non-Random File Size Compression Comparisons

Since the file size differences were not constant for compressed utilities, comparisons between the algorithms were done via the standard deviations. The standard deviation for non-compression was always 0 because there was no variation in the file size differences for the different input file sizes for each encryption scenario. The standard deviations between the file size differences for RSA were similar to ELG-E, while the file size differences for 3DES, AES, BlowFish, CAST5, and TwoFish were very similar as well. Because of the similarities between RSA and Elg-E and between the symmetric algorithms, the next two graphs only show the compression comparisons for RSA and 3DES for randomly-generated files.

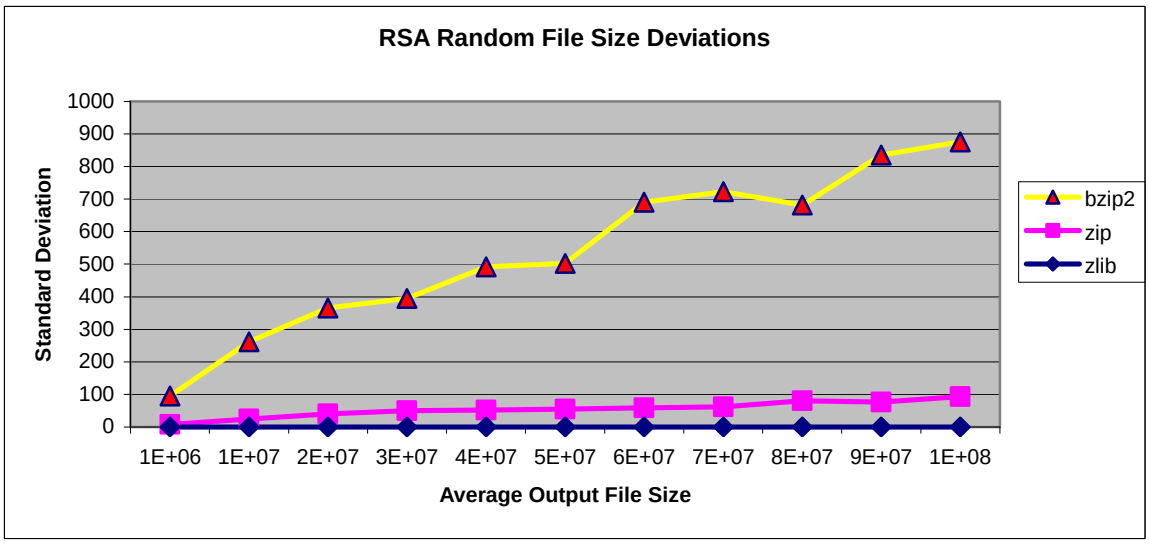

Figure 14. RSA Random File Size Difference Standard Deviation

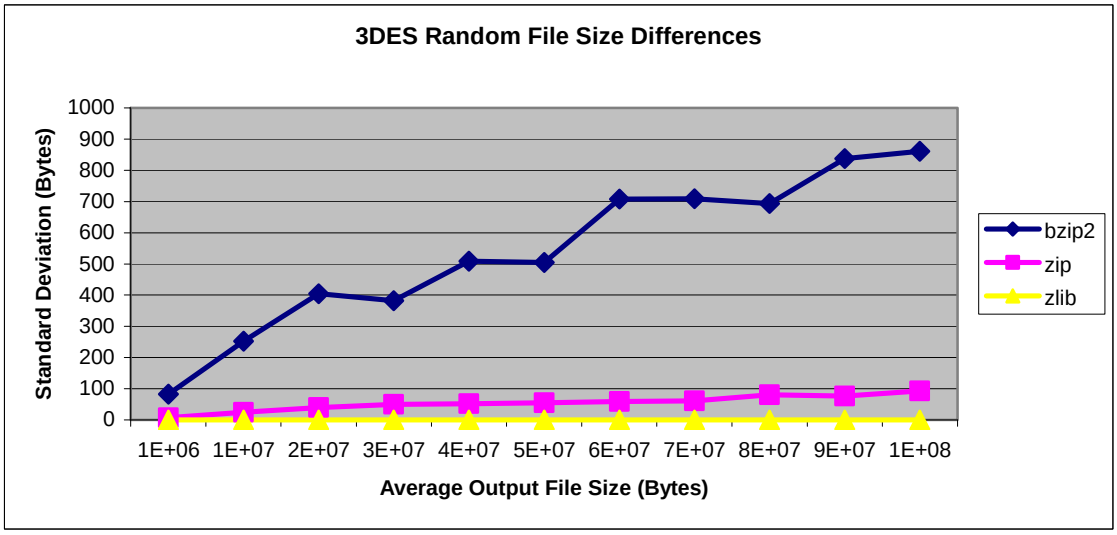

Figure 15. 3DES Random File Size Difference Standard Deviation

As the above graphs illustrate, for all algorithms, zlib's standard deviation for the 30 different runs for each scenario was zero or very close to zero, consequently, zlib was more predictable. The standard deviation for zip was also predictable because its'

standard deviation remained nearly constant at under 100 while the standard deviation for bzip2 increased mostly for each size increase.

When the files were not random, the standard deviations displayed more characteristic differences. Primarily, the standard deviations for zip and zlib were very close to 0 (more predictable), while the bzip2's standard deviation showed increases and decreases. The following input files sizes were used for the non-random testing:

- 1162313
- 10437672
- 20576925
- 31014597

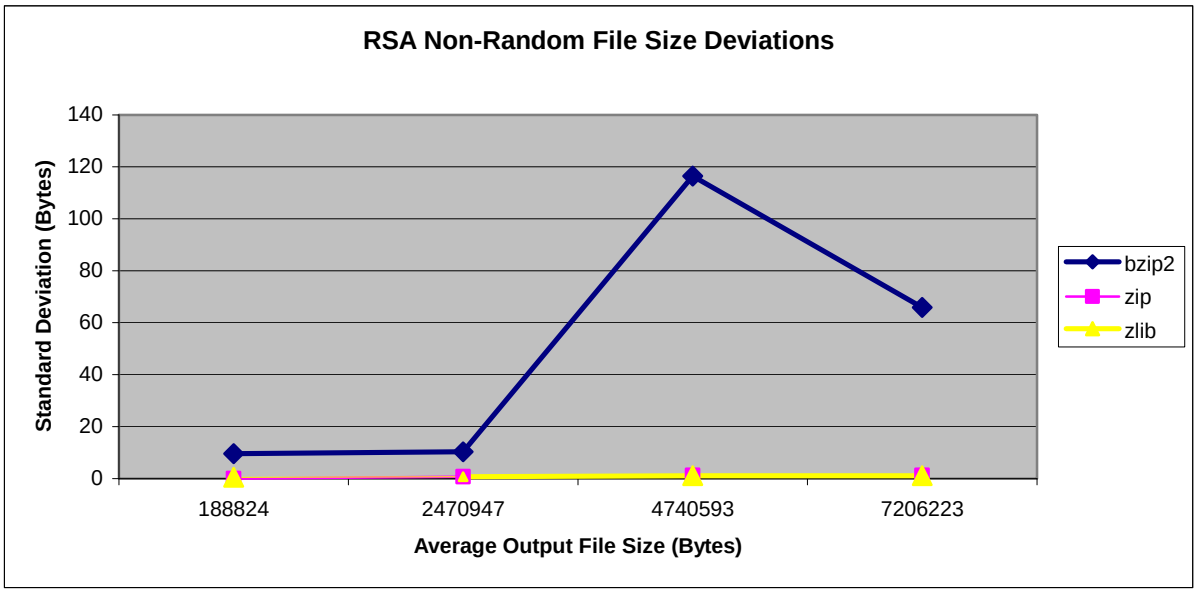

Figure 16. Non-Random RSA File Size Difference Standard Deviations

Not only did bzip2 show non-predictability within RSA, it also showed non-predictability with the other algorithms as Figure 17 illustrates.

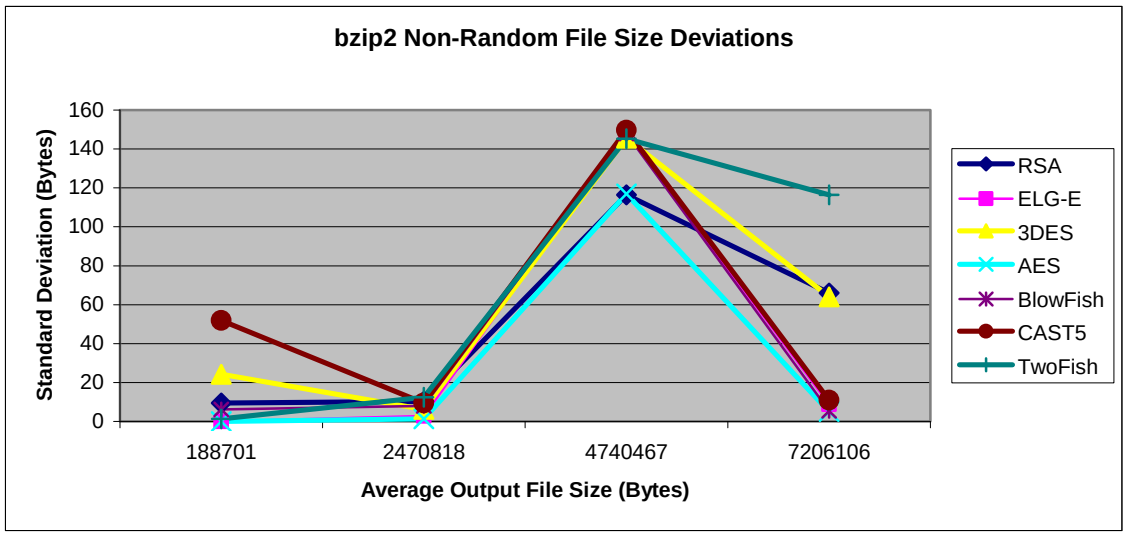

Figure 17. Bzip2 Non-Random File Size Difference Deviations

All of the algorithms had a significant spike for the average output file size of approximately 4740400B or 4.7MB (which corresponds to input file size 20576925B or 20.6MB). This could be attributed to the memory of the computer used. Depending upon the virtual memory available, the compressions under the available virtual memory could fit into memory but greater than this would require swapping files out of primary memory for compressing. In order to fit completely into memory, the size of the file would have to be less than the available virtual memory (other data resides in memory besides what is needed for encryption and compression). The CAST5 had the most significant disparities beginning with 51 Bytes standard deviation for the 188701 (.2MB) output file size and ending with an 11 Byte standard deviation for the greater output file size (7206106 (7.2MB)). AES had the smallest standard deviations for the different output file sizes. TwoFish had the highest discrepancy for the output file size of approximately 7206000 (7.2MB).

4.2.2 Encryption and Decryption Results.

The encryption time for each scenario was measured from the beginning of the encryption process until the end of the encrypting process. Overall the symmetric key algorithms required less time to encrypt the files than the public key algorithms.

4.2.2.1 Encryption and Decryption Results for Non-Compressed Files.

Although 3DES, CAST5, and BlowFish were better on file differences for random files (see Figure 10 from Section 4.2.1.1), they did not have better encryption run times. In fact, 3DES had the longest running time as shown in Figures 18 and 19.

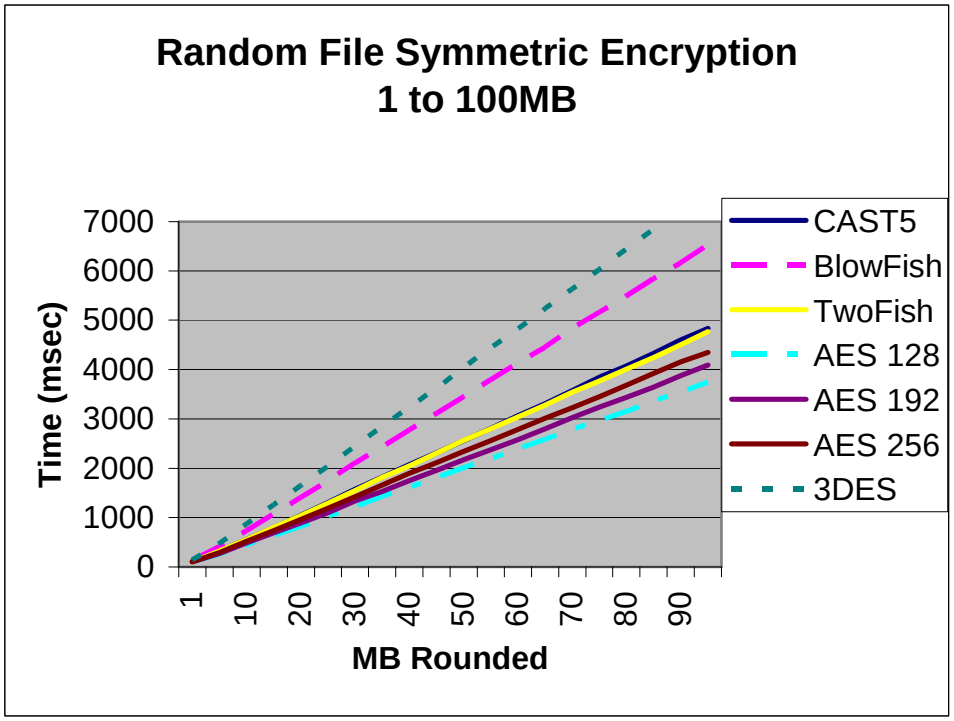

Figure 18. Symmetric Encryption for Random Files to 100MB

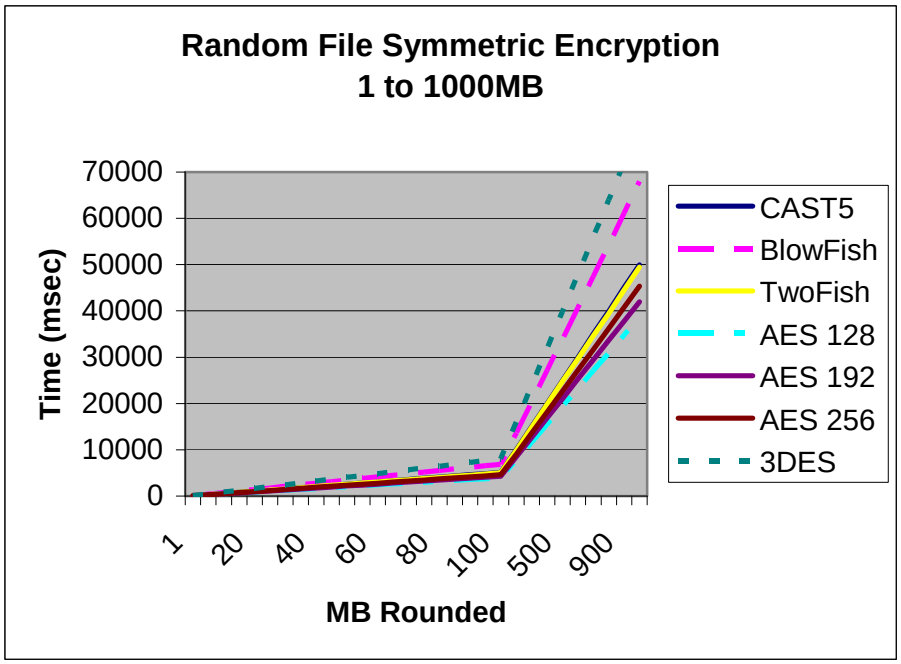

Figure 19. Symmetric Encryption for Random Files to 100MB

In addition, AES with 128 bit encryption had the fastest encryption times. In fact, AES with larger key sizes of 192 and 256, had faster encryption times than the 128 bit TwoFish, BlowFish, CAST5, and 3DES. TwoFish was only slightly better than CAST5 for encryption times. All of the symmetric algorithms increased encryption times with increase input file size. Table 11 shows a comparison of the symmetric key algorithms.

Table 11. Symmetric Encryption (msec) for Random File Sizes

Input (MB)	CAST5	BlowFish	TwoFish	AES 128	AES 192	AES 256	3DES
1	106.667	128.333	105.667	96	101.333	103	141.667
5	318	411.333	322.333	259	274	290	474
10	557	738.367	554.667	449	482.333	514.333	868.333
15	797.167	1080.37	802.9	631	688.7	736.333	1251.67
20	1048.67	1408.37	1039.33	822.2	892.667	963.7	1650.33
25	1303.37	1747	1292	1026	1097.67	1189.67	2044
30	1575.97	2106.67	1549.07	1218	1337.67	1423	2444.33
35	1819.67	2439.33	1814.3	1428	1531.07	1663.67	2843.03
40	2074.67	2781.33	2051	1613.9	1749.6	1899.03	3236
45	2314.77	3113.43	2310.37	1809.3	1957.33	2115.33	3628.67
50	2561.63	3452.67	2567.33	2009	2173.33	2339.77	4041
55	2802.67	3794.43	2800.7	2192	2372.83	2560.6	4443.1
60	3066.4	4139	3035	2389.47	2577.97	2784.67	4816
65	3311.6	4456.37	3275.4	2584.43	2798.33	3011.03	5235.9
70	3572	4837.33	3542.33	2783.03	3023.33	3228.33	5627.33
75	3848.87	5162.67	3768.67	2965.33	3238.77	3450.33	6030.7
80	4074.13	5490.9	4005.53	3145.43	3439.63	3687.33	6441
85	4331.67	5840	4246	3363.07	3646.67	3920.5	6838.2
90	4600.67	6165.37	4504.03	3550.33	3878.03	4158.57	7230.33
95	4832.43	6529.77	4770.83	3742.73	4093.63	4350.4	7626.63
100	5096.5	6839.47	5000.4	3923.53	4289.03	4584.73	8020.43
200	10066.8	13628.7	9907.77	7808.03	8461.03	9054	15960.3
300	15031.7	20423.6	14865.8	11625.1	12689.4	13630.5	23955.3
400	19987.5	27164.8	19842.2	15510.5	16875.9	18144	31890.1
500	24965.5	33907.2	24772.1	19403.3	21040	22655	39833
600	29935.3	40620.6	29650.1	23251.9	25260.2	27165.9	47828.3
700	34987.5	47430.7	34595.8	27073.1	29429.5	31648.9	55888.9
800	39982.7	54270	39445.7	30911.8	33657.4	36230.4	64025
900	45072.8	61130.2	44541.1	34801.4	37779.6	40814.2	71994.6
1000	49980.9	67937.3	49458	38589.3	41980.8	45302.9	79918.2

For symmetric decryption results, the same order for encryption speed was seen in decryption speed. AES 128 had the fastest decryption times, followed by AES 192, AES 256, TwoFish, CAST5, BlowFish, and lastly 3DES.

For public encryption algorithms, RSA performed better than ElGamal for higher key sizes. ElGamal had faster encryption times for key sizes 1024 and 1280 (key size 768 was only tested with ElGamal. Once the key size increased above 1280, RSA rose slower than ElGamal and remained rather steady for each input file size across the key size range. Figure 20 shows the steadiness of RSA encryption time as opposed to ElGamal's increasing encryption times for a 10MB file with key sizes from 1024 to 4096.

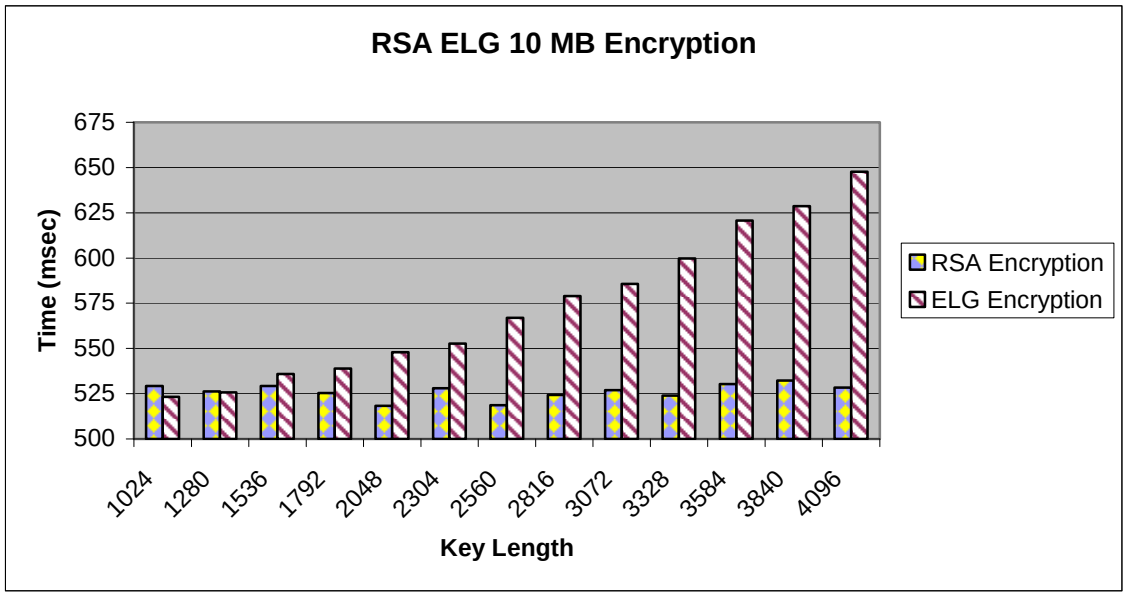

Figure 20. RSA and Elg-E 10MB Encryption Time

For RSA's encryption time increased as the key size increased except for key size 2048. At this key size, the encryption time was less than the encryption time for key size 1024.

A comparison of decryption times for the public key algorithms, show that RSA has a slightly better decryption time overall than Elg-E above key size 1280. Figure 21 shows a comparison of the decryption times for RSA and Elg-E.

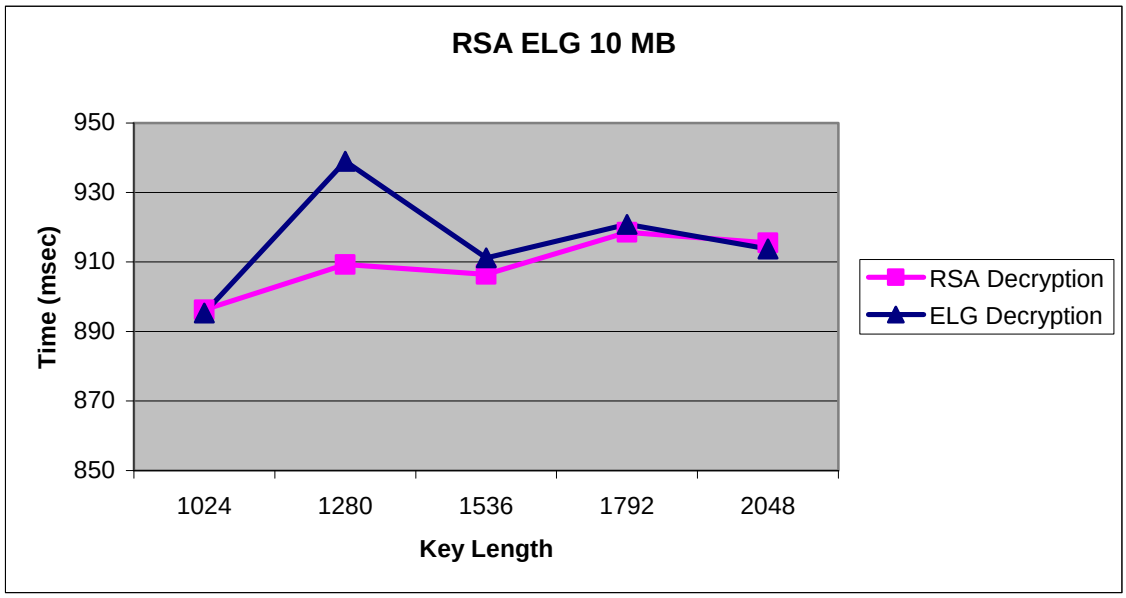

Figure 21. RSA vs. Elg-E Decryption Time Comparisons

4.2.2.2 Encryption Results for Compressed Files.

The zip compression utility yielded the fastest encryption time for random and non-random files followed by zlib. Bzip2 had the slowest encryption time. A sample of the data collected is displayed in figures few of the graphs are displayed below:

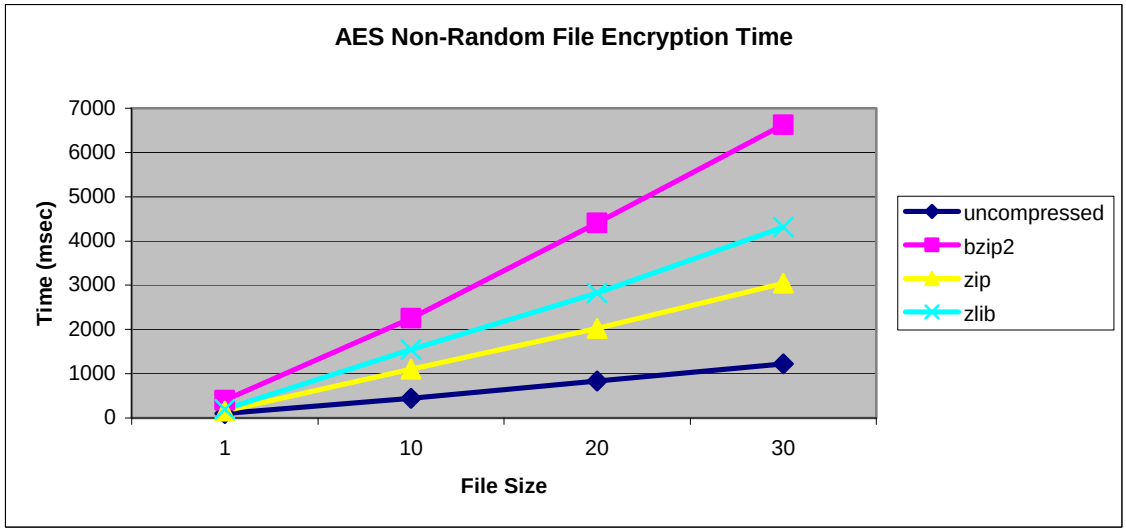

Figure 22. AES Non-Random File Compressed Comparisons

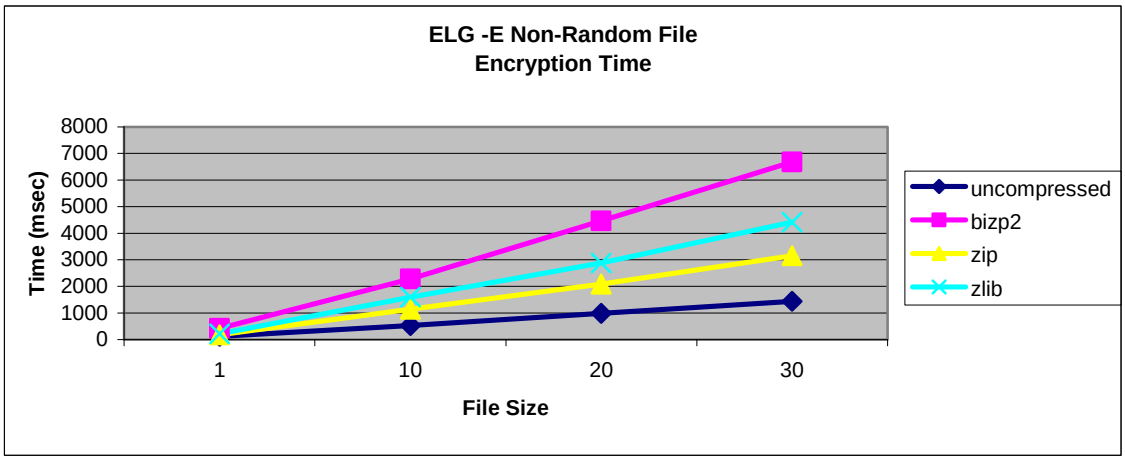

Figure 23. Elg-E Non-Random Compression Comparisons

4.2.3 Overall Analysis of Algorithms.

The data gathered from the gpgTester executions was used as input for the MatLab interpolator and controller. Additionally, the data analysis from this section combined with the information from the references listed in the bibliography was used to create the security levels and the performance levels for Controller One and Controller Three. Because of the requirement within Controller Three for only three algorithms per security and performance level for binary programming, Controller One and Controller Three had different encryption schemes for the security and performance levels.

The security levels were divided into five levels, high (5) to low (1) while the performance levels ranged from high (3) to low (1). Table 12 lists the encryption choices for each security and performance levels for Controller One whereas Table 13 lists the security and performance levels for Controller Three (3 encryption choices per security/performance level).

Table 12. Security and Performance Levels for Controller One

Security Level	Performance Level	Encryption Algorithm	Key Size
1	1	RSA	1024, 1280
1	2	ElGamal	768, 1024, 1280
1	3	3DES, BlowFish	128
2	1	ElGamal	1536, 1792
2	2	RSA	1536, 1792
2	3	CAST5	128
3	1	ElGamal	2048, 2304, 2560, 2816
3	2	RSA	2048, 2304, 2560, 2816
3	3	AES, TwoFish	128
4	1	ElGamal	3072, 3328, 3584, 3840
4	2	RSA	3072, 3328, 3584, 3840
4	3	AES	192
5	1	ElGamal	4096
5	2	RSA	4096
5	3	AES	256

Table 13. Security and Performance Levels for Controller Three

Security Level	Performance Level	Encryption and Key Size
1	1	RSA 1280 and 1536; Elg-E 1280
1	2	Elg-E 768 and 1024; RSA 1024
1	3	3DES, BlowFish, CAST 5
2	1	Elg-E 1536 and 1792; RSA 2304
2	2	RSA 1792 and 2048; Elg-E 2048
2	3	CAST5, 3DES, BlowFish
3	1	Elg-E 2816 and 3072; RSA 3072
3	2	RSA 2816 and 2560; Elg-E 2560
3	3	AES, TwoFish, CAST5
4	1	Elg-E 3840 and 3584; RSA 3840
4	2	RSA 3328 and 3584; Elg-E 3328
4	3	AES 192, TwoFish, CAST5
5	1	Elg-E 3840 and 4096; RSA 4096
5	2	Elg-E 3584; RSA 3840 and 3584
5	3	AES 256 and 192; TwoFish

4.3 MatLab Interpolation (Step Two)

The results from this step was used as input into the MatLab controllers. The data from the gpgTester was compiled into one MatLab file, testing1.m. This file was used to create formatted matrices for the other MatLab files to use. There were 1363 lines of data compiled into testing1.m. Each line represent one averaged encryption scenario from the gpgTester. A sample listing of the code is shown below (note: the first line is included only for reference):

Table 14. Testing1.m Sample File Output

A	B	C	D	E	F	G	H	I
0	1	0	128	1048576	1048673	97	137.333	936.900
0	5	3	1024	90177536	90216419	38883	11726.600	7813.870

The random file category, A, was a 0 for a randomly generated file and a 1 for a non-randomly generated file. For the algorithm category (B), AES was 0, TwoFish was 1, CAST5 was 2, 3DES was 3, BlowFish was 4, ElGamal was 5, and RSA was 6. For the compression category (C), non-compressed was 0, bzip2 was 1, zip was 2, and zlib was referred to by a 3. The other categories were key size (D), input file size (E), encrypted output file size (F), file size difference (G), encryption times (H), and decryption times (I).

During Step Two, cubic splining was performed to interpolate input file sizes with the data gathered from Step One. For any input file size (within the tested data ranges), an interpolated output was produced. This interpolated data, when inserted into the

original data, would produce a smooth line between the data points as shown in Figures 24 and 25.

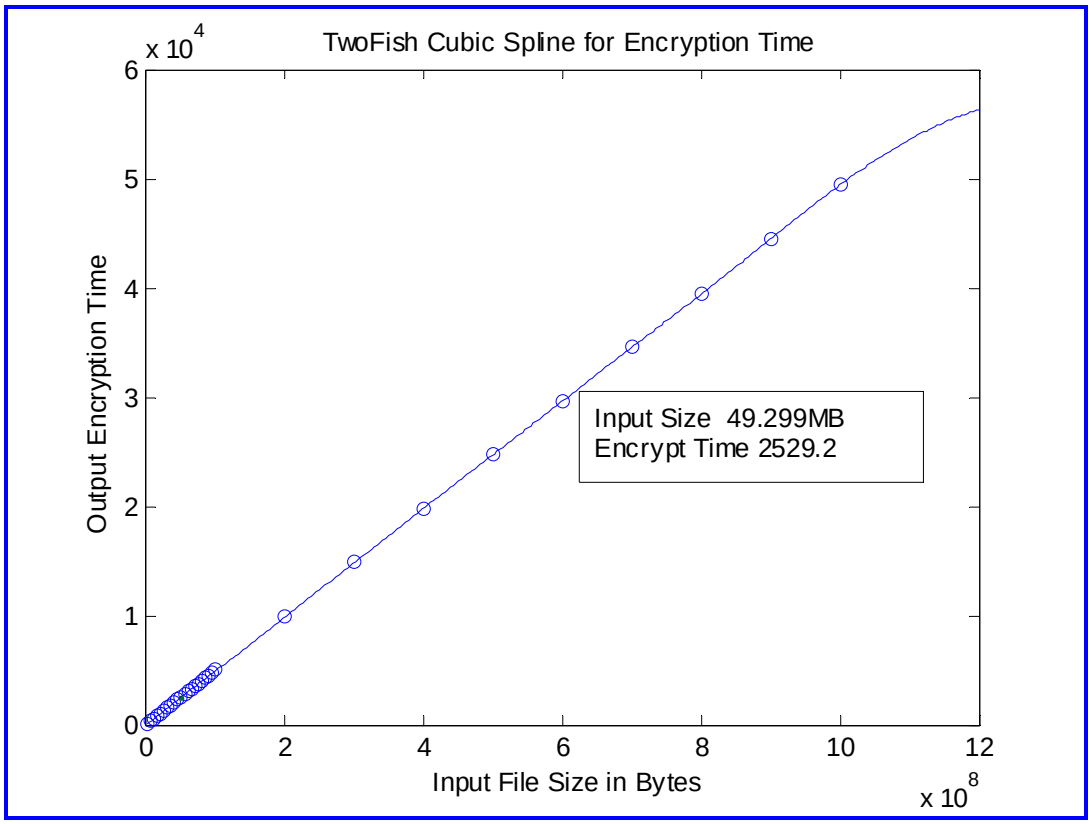

Figure 24. TwoFish Cubic Spline for Encryption

Figure 24 shows the best curve fit for determining the encryption time for any given input using TwoFish with 128 bit encryption. In the graph, the input value was 49.299MB. This value is interpolated with the TwoFish results from the gpgTester. From the gpgTester, the following data is closest to the input file size: 45.089MB with 2310.37 encryption time and 50.070MB with encryption time of 2567.33. The cubic spline interpolation outputted a 2529.2 encryption time for the 49.299MB file which places the encryption time between 45.089MB and 50.070MB but closer to the latter as it should.

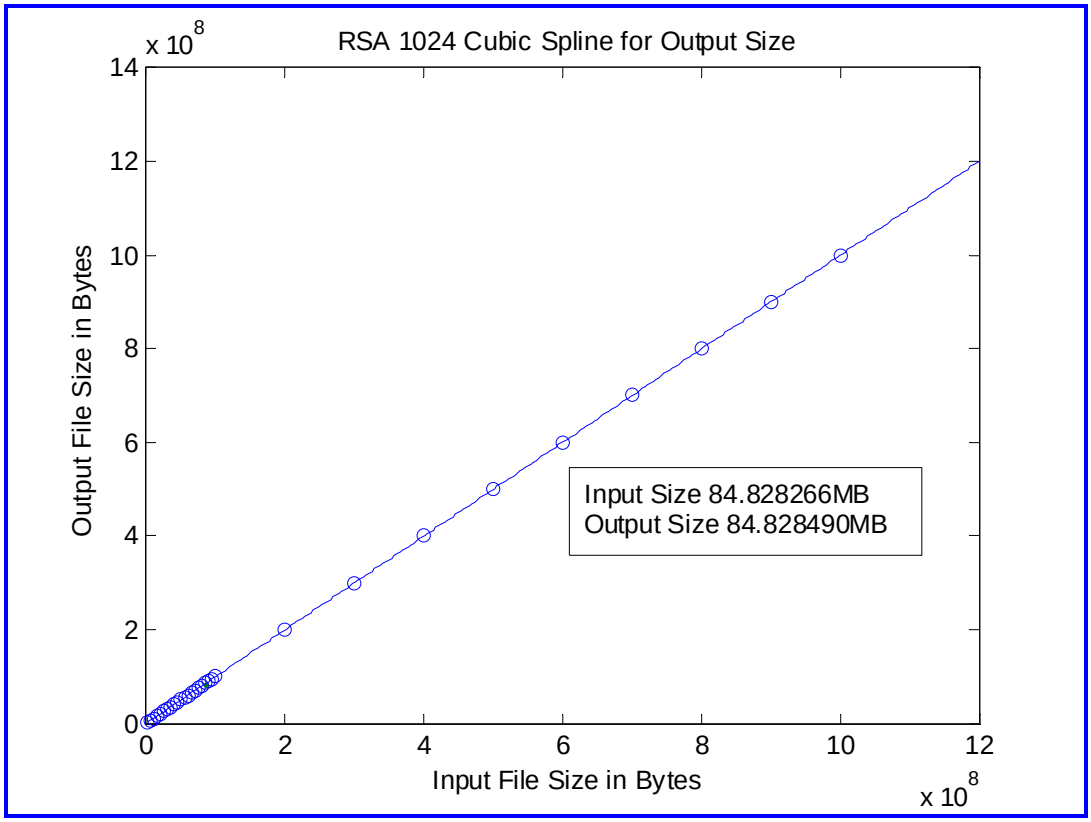

Figure 25. RSA Cubic Spline for Output File Size

A sample of the cubic spline for the output file size is shown in Figure 25 for RSA with 1024 bit encryption. The input file size was 84.828266MB while the output file size was 84.82849MB. The file size difference was 224 which is the same file size difference as seen with the gpgTester with RSA at 1024 bits.

4.4 MatLab Controller (Step Three)

The interpolation done in Step Two was fed into the MatLab controllers for optimizing the commodities that could be sent. The controllers each had unique data results based upon the function of the controller.

4.4.1 Random Commodity Analysis.

Controller One and Controller Two employed random number generators for commodity file size and priority. Random number generators allowed for a wider range of values to be tested without being subject to researcher's preferences. A gamma distribution was used to generate the commodity file sizes while a uniform distribution was used for the priorities.

A sample histogram of the gamma distribution for file sizes is shown in the Figure 26.

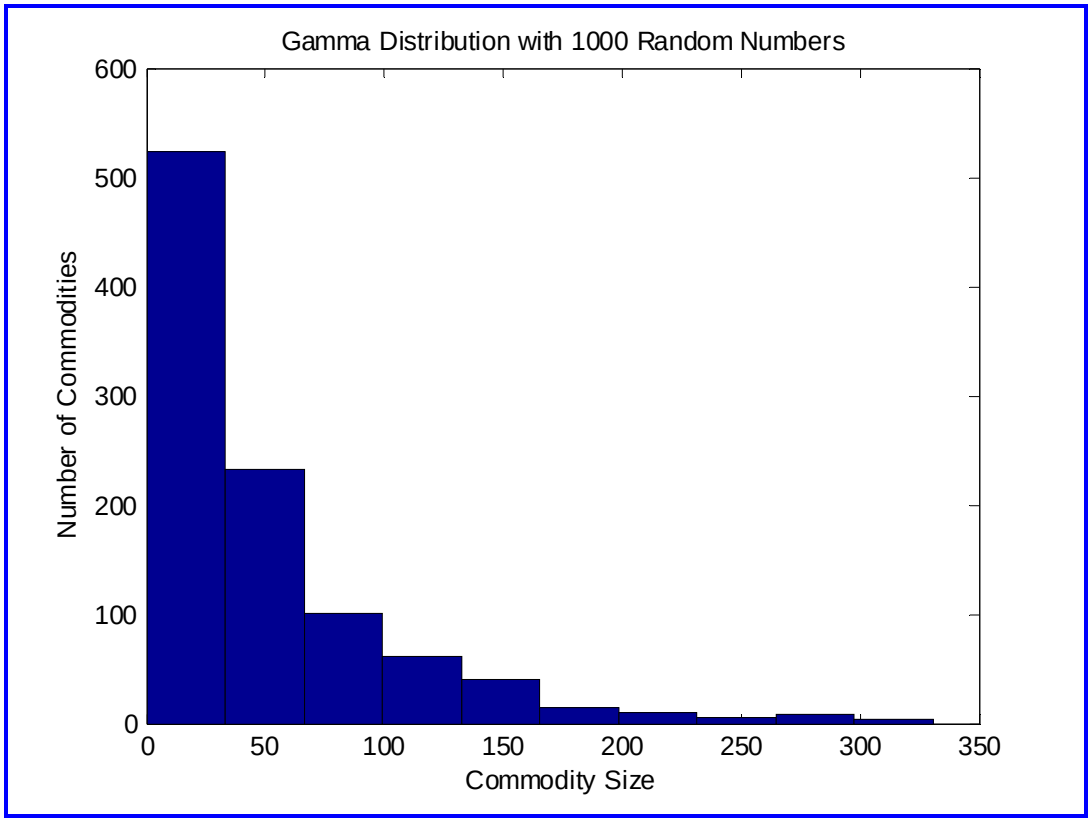

Figure 26. Gamma Distribution for File Sizes

The histogram showed that most of the random numbers for commodity file sizes fell between 0 and 150MB. Several different sample histograms were created but show basically the same information as illustrated in Figure 26.

For testing purposes, the input file sizes were restricted to 1MB to 100MB (very rarely are files greater than 100MB and most are under 30 MB). To enforce this restriction, only numbers between 1MB and 100MB were allowed for the file sizes.

The next histogram (Figure 27) shows a sample modified gamma distribution with the file size limitations. The shapes of the histogram reveal the reverse exponential graph similar to the histograms from the non-modified gamma distribution.

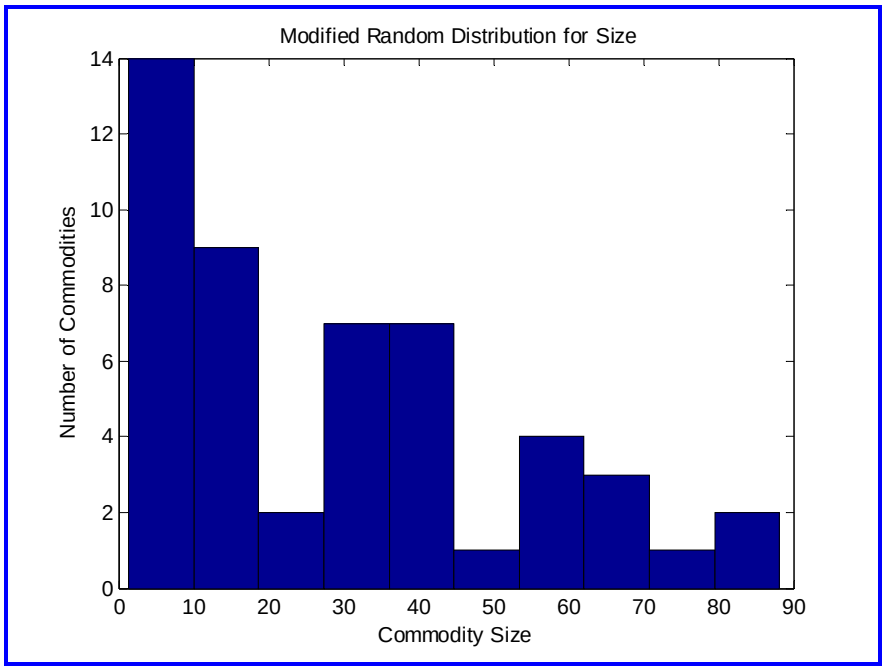

Figure 27. Modified Gamma Distribution for File Size

Although the shapes of Figure 29 and 30 are similar to the non-modified gamma distribution, the uniformity of the bars are not as consistent as Figure 27. This is because of the file size restriction. If a randomly generated number via gamma distribution was

above 100MB or below 1MB, then this number was not used and another number was generated instead.

The histogram (Figure 28) for the commodities' priorities (the goodness) showed a uniform random distribution and not a reverse exponential increase (each priority from 1 to 100 have an equal probability of occurring).

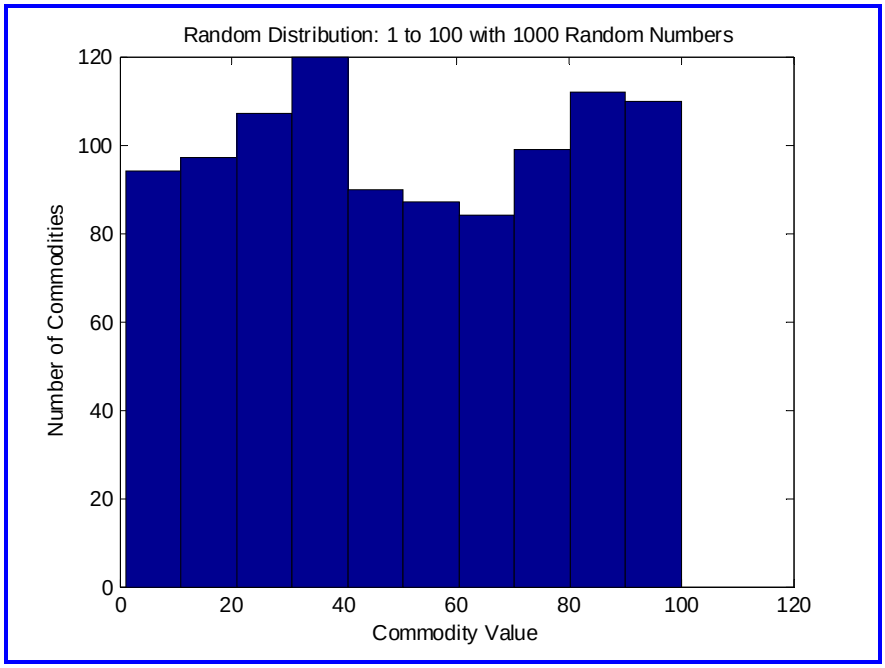

Figure 28. Uniform Distribution of Commodity Priorities

The commodity matrix was a two column matrix with input size and value (priority). Once the controller obtained the commodity matrix either via random generators or input files, it proceeded to optimize the data.

4.4.2 Controller One Analysis.

Controller One used the concept of security levels and performance levels to optimize the data. Table 12 from Section 4.2.3 shows the encryption schemes per security and performance level.

The controller maximized the commodity by first sorting the commodities from high to low and then only choosing the encryption scheme (determined by the security and performance level) which maximized the commodities (i.e. the aggregated encryption time and encrypted output file size did not exceed the available CPU and available bandwidth).

The output from this optimization was a four column matrix with input size, value, output encrypted file size and encryption time. In addition, the sum of the priorities (the total goodness), total bandwidth required, and the total CPU required are also written to the output file. Table 15 shows a sample output file from Controller One.

Table 15. Sample Output from Controller One

1	***120000000***	***3000***	***20***	***4***	***3***
2	***437***	***64395928.920***	***2984.092***		
3	77241394.300	30			
4	6486429.104	84			
5	92371565.296	78			
6	4709272.829	40			
7	46658998.933	40			
8	90862156.535	52			
9	31074677.131	97			
10	88939001.923	90			
11	95480951.768	67			
12	48285980.266	31			
13	57122688.347	37			
14	20101790.204	77			
15	20802228.353	12			
16	4700810.855	34			
17	48685609.267	71			
18	2756101.424	84			
19	29118058.790	3			
20	80469714.869	59			
21	3976446.057	95			
22	14607527.715	35			
23	***0***	***0***	***0***	***192***	
24	31074677.131	97	31074774.131	1376.479	
25	3976446.057	95	3976543.057	221.327	
26	6486429.104	84	6486526.104	325.975	
27	2756101.424	84	2756198.424	170.951	
28	20101790.204	77	20101887.204	889.359	

Line 1, from the above table, show the available bandwidth, available CPU, number of commodities to be randomly generated, the security level, and the performance level, respectively. Line 2 list the total goodness of the commodities chosen, the amount of bandwidth that will be required, and the amount of CPU needed to encrypt the commodities. Lines 3 thru 22 list the 20 commodities (file size and priority) that were randomly generated. Line 23 show the encryption scheme that yielded the

maximum number of commodities. Lines 24 thru 28 list the commodities that can be sent including the output encrypted file size and encryption time.

4.4.3 Controller Two Analysis

Controller Two introduced binary integer programming to optimize the commodities. It did not utilize the concept of security and performance levels, rather it used seven encryption schemes per commodity. These seven encryption schemes were mutually exclusive (at most one encryption scheme was chosen by the solver).

Initially, the testing tried to execute binary integer programming via the bintprog function installed within MatLab, however, warning messages were periodically received stating that the results may be inaccurate.

Because of the unpredictability of the bintprog function, a different binary integer solver was researched and incorporated into MatLab. The new solver, glpk, used a mex interface to run within MatLab. The same testing done with bintprog was used to test the accuracy of glpk. Through the testing, glpk did not exhibit any erratic behavior exhibited by bintprog.

The outputted data from the glpk function included the *x* vector which represented the commodities that could be sent (see Section 3.4.4 for the explanation of the vectors and matrices used for binary integer programming). During the verification process, the *x* variables were analyzed to ensure that mutual exclusion was maintained and that the available CPU and available bandwidth requirement was fulfilled. To verify the accuracy of the outputted data (encryption times, encrypted file sizes, mutual exclusion of encryption algorithms) from glpk, hand calculations were used. The hand calculations

reviewed each *x* variable and aggregated the encryption time and encrypted file size for each variable equal to 1.

The controller read (from an input file) the available bandwidth, available CPU, and the number of commodities to be generated. The output was sent to a file. It included the commodities that could be sent. The *x* vector, the status of the solver, the total goodness, and the encryption and bandwidth required.

The number of commodities tested ranged from 4 to 100 commodities. Each time the solver ran, a different set of commodities was used. In addition, the available bandwidth and available CPU varied as well. During this testing, the solver slowed down as more commodities were introduced. A problem occurred when Controller Two attempted to execute the input lines from a file within a for loop (each input line included the available CPU, available bandwidth, and number of commodities to generate). After approximately 57 commodities (sometimes less depending on the commodities generated), MatLab seemed to freeze. However, to work around this problem, the input lines were fed into the controller one at a time (instead of reading 10 lines of input and executing in a for loop, it read one line of input and then the program was called again for the next nine lines of input). By doing this work around, the solver was able to solve for 100 commodities (100 was the limit for this research, however, to verify that this was not the limit for glpk, higher commodities were tested successfully).

Each line of input was run five times to verify that the solver worked and to view the range of solutions solved. For analysis to stabilize the input commodities, the same

commodities (initially generated randomly) were used for the testing of 5 to 100 commodities, with the same available bandwidth and CPU.

A sample output for five commodities with an input of 70MB available bandwidth and 3000 seconds for the available CPU yielded the results listed in Table 16.

Table 16. Sample Output File for Controller Two

1	**Input line**	**70000000**	**3000**	**5**
2	Sum Totals	225	62350813	2997
3	Commodity 1	2063745.786	84	
4	Commodity 2	58624370.173	81	
5	Commodity 3	1662180.473	60	
6	Commodity 4	53546860.907	62	
7	Commodity 5	31546633.470	65	
8	*x* vector			
9	*Status*	5		
10	*Time*	0		
11	*memory*	2954.688		
12	*Commodities to send*			

Row 1 is the input line read into the Controller for five commodities. Row 2 is the sum totals for the goodness (the priorities), encrypted file size, and encryption time

for the commodities chosen. Row 3 to Row 7 refer to the five commodities requested (file size and priority). Row 8 is the *x* vector which holds the values of the optimum decision variables for the solver. Row 9 is the status of the optimization. Line 10 refers to the time in seconds for the solved solution. Row 11 refers to the amount of memory required by the solver in Kilobytes. Lastly, Row 12 refers to the commodities that can be sent based upon the *x* vector.

The *x* vector for the sample (every other commodity shaded) was 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0. There were three ones for the *x* variable corresponding to commodity 1, 2 and 3. Commodity 1 used encryption scheme 3, commodity 2 used encryption scheme 6, while Commodity 3 used encryption scheme 5.

The status field of the output refers to the state of the optimization after the solver terminated. There were six general status codes plus error codes possible:

- 1 = solution was undefined
- 2 = solution was feasible
- 3 = solution was infeasible
- 4 = no feasible solution existed
- 5 = solution was optimal
- 6 = solution was unbounded
- 101 – 110 = error code (10 different error codes)

The commodities to be sent was written in matrix form where each commodity had nine different columns. Table 17 shows the actual commodities to be sent for the above sample.

Table 17. Sample Commodities to Send for Controller Two

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
2063745.786	84	3	0	4	0	128	2063810.786	199.098
58624370.173	81	6	0	5	0	1024	58624725.173	2657.994
1662180.473	60	5	0	1	0	128	1662277.473	140.382

Only 3 commodities out of the 5 requested were chosen for transmission. Columns 1 and 2 (from Table 17) refer to the original commodities. Column 3 refers to the encryption scheme number. Note, the numbers are the same as the *x* variables chosen (3, 6, and 5). Columns 4 to 7 refer to the actual encryption scheme. Column 4 referred to whether or not the interpolation should use the random file data (from gpgTester). Column 5 referred to the encryption type (AES was 0, TwoFish was 1, CAST5 was 2, 3DES was 3, BlowFish was 4, Elg-E was 5, and RSA was 6). Column 6 referred to whether or not compression should be used. Column 7 referred to the key size for the algorithm. Column 8 lists the encrypted file size. Finally, Column 9 list the encryption time.

Column 8 was added together to determine the amount of bandwidth the three commodities would require and likewise for Column 9 to determine the amount of CPU.

A compilation of another sample using 120MB as the available bandwidth and 5000 as the available CPU is shown in Table 18.

Table 18. Compilation of Controller Two Output

# of Commodities	# to Send	Time for Solver	Memory for Solver	Goodness Total	Encrypted File Size Total	Encryption Time Total
5	4	0	2999.316	290	93897189.9	4987.178
10	6	0	2999.316	409	107047826.6	4981.642
20	8	0	2999.316	580	116931260.1	4995.799
30	12	0	2999.316	783	110070128.9	4999.14
40	14	0	2999.316	899	106841591.5	4990.108
50	15	1	2999.316	1023	107961098.6	4998.682
60	16	2	2999.316	1049	107160660.9	4998.986
70	16	2	2999.316	1049	107160660.9	4998.986
80	16	3	2999.316	1082	106587650.4	4999.787
90	16	4	3091.422	1082	106587650.4	4999.787
100	17	3	3091.422	1145	104502014.9	4989.059

The above chart shows that the solver tries to maximize the encryption time and the encrypted file size while still attempting to maximize the number of commodities that can be sent. The greatest number of commodities were for 17 commodities for an initial 100 commodities. The memory requirements for 100 commodities were very similar to the requirements for 5 initial commodities. The time for the solver only increased slowly after 50 initial commodities, however, the time decreased by 1 for 100 commodities. As expected the goodness total increased with an increase in input commodities. Note, this

data was reflective of other test runs with different initial input commodities. The characteristics were the same for solver time (between 0 and 4 observed), solver memory requirements (between 90 to 3100 was observed), and goodness total. The number of commodities sent, however, was dependent upon the randomly generated commodity file sizes and priorities.

4.4.4 Controller Three Analysis

Controller Three merged the security and performance levels with the binary integer solver to determine which commodities could be transmitted. This controller did not use randomly generated commodities but rather the commodities from an input file (similar to NS-2 output files).

Each commodity input line included the input file size, priority, security level, and performance level. Each of the security and performance levels were translated into three available encryption schemes as listed in Table 13 in Section 4.2.3. These three encryption schemes became the basis for the mutual exclusions for the constraints matrix, *A* (See Section 3.4.5 for an explanation of binary integer programming including the constraints matrix).

The output file included the same items as Controller Two plus the original security and performance level for the commodity. For a sample of the type of output captured, see Table 17 from Section 4.4.3.

4.5 Analysis of NS-2 Simulations (Step Four)

The simulations from NS-2 provided the commodities, the available bandwidth, available CPU, and the security and performance level for each commodity. The format

was the same format as the Controller Three, however, for simulation purposes, the output did not include the *x* variable output.

To actually run the simulations, input and output files were used to transfer data from MatLab to NS-2. The NS-2 simulations were run via the Hybrid Agent for Network Control, a tool created by John Pecarina in his research of agent based frameworks (Pecarina, 2008).

A sample line from an output file is:

1 10000000.000 60 3 1 0 5 0 3072 10000867.000000000 575.648

The first five values were the original values sent via the input file. These values list the commodity flow number, the commodity file size, and priority followed by the security and performance levels. The Controller Four (encryptFitter) determined the maximum commodities that could be sent and added six additional values to each line of the commodity (that was chosen). These additional values were the four values for the encryption scheme (randomness, algorithm, compression, and key size), the required encrypted file size, and the encryption time.

4.6 Comparison of Binary Solver and Non-Binary Solver

There were two different methods for optimizing the number of commodities that could be transmitted at a node, a non-binary solver (Controller One) and a binary solver (Controllers Two, Three, and Four). The comparison was between Controller One and Controller Two because the other controllers build on Controller Two. In order to compare the output from the different types of solvers, the input data had to be the same,

therefore, the random generation of the commodities were frozen, i.e., the same commodities were used for each controller comparison. In addition, the security and performance levels were not used for Controller One. Both Controllers used the same seven encryption schemes.

Overall Controller 2 using binary integer programming allowed more commodities for transmission and higher sums for the priorities. Table 19 shows a sample comparison.

Table 19. Sample Comparison of Controllers

	# of Initial Commodities	**Available BW MB**	**Available CPU (Sec)**	**Total Goodness**	**# of Commodities to Transmit**
Controller 1	5	70	3000	225	3
Controller 2	5	70	3000	225	3
Controller 1	20	120	5000	508	6
Controller 2	20	120	5000	580	8
Controller 1	40	120	5000	630	7
Controller 2	40	120	5000	899	14
Controller1	100	120	5000	580	7
Controller 2	100	120	5000	1145	17

The controllers using binary integer solver was able to maximize the output better than the non-binary integer solver. Controller 1 became more inefficient as the number

of commodities increased whereas Controller 2 was able to increase efficiency even when the bandwidth and CPU remained constant as the commodity number increased.

V. Conclusions and Recommendations

5.1 Summary of Research

Dialable cryptography for wireless networks provides users with the means to control their security requirements especially in a dynamically changing environment.

The research used two public key (RSA and ElGamal) and five symmetric key (AES, 3DES, TwoFish, BlowFish, and CAST5) algorithms. Although public key encryption is normally not used to encrypt a file, it was used in the testing. Periodically, public keys could be used to encrypt files especially since the encryption keys are public whereas for symmetric key algorithms, the keys must be shared prior to encrypting.

With any encryption algorithm, there are weaknesses within the encryption scheme and some encryption algorithms are considered better than others. Regardless, a "cryptosystem is secure if the best known attack requires as much work as an exhaustive key search. By this definition, a secure cryptosystem with a small number of keys could be easier to break than an insecure cryptosystem with a large number of keys." (Stamp, 2006). In other words, larger key sizes can increase the security of public and symmetric key algorithms.

By dividing the encryption algorithms into different security levels and performance levels, users do not need to know the actual encryption algorithm but rather their estimate of the security required and of the performance requirements. Furthermore, the MatLab controller conceals the binary integer programming (optimization) and data interpolation from the user.

The analysis of the gpgTester was instrumental in the data collected from the interpolation and optimization steps. Because of this interdependency, a thorough analysis of the gpgTester data was required prior to completing the other steps.

The interpolation of the data from the gpgTester was used by the MatLab controller for optimizing. The data interpolation was limited to encryption and primarily to non-compressed algorithms. For future research, the initial data must be varied enough to support interpolation of different requirements including compression.

The controllers provided three different methods for optimizing (security and performance levels, binary integer programming, and a combination of the two) the commodities. These three different methods can be used to meet different user's objectives for the commodities.

The fourth controller was basically the same as the third controller but was used to integrate into a network agent (HANC) via NS-2.

A major conclusion from the analysis of the controllers is that the binary integer solver maximized the number of commodities that can be sent and maximized the sum of the commodities' priorities.

5.2 Future Research

The research conducted focused on the objective of creating a controller to determine encryption algorithms based upon security and performance levels for the transmission of the maximum number of commodities within the available bandwidth and CPU. The research assumed that the bandwidth was correct as inputted, future research could delve into actually determining the bandwidth for wireless networks. In

addition, more work would be needed into determining what the security and performance level dials should be. This would require researching security policies and incorporating these policies into the performance and security levels of the encryption algorithms.

Although only seven encryption algorithms were used, future research could use other encryption algorithms to encrypt packets with a size range of under 1MB. The research only investigated file sizes of over 1MB. In addition, another research possibility would be to change the cryptographic algorithm for different blocks within a file similar to the AdaptCrypt created by Manzanares (Manzanares, Camara et al., 2005). Different encryption schemes for one file or packet would make it more secure during transmission.

Regardless of different encryption algorithms or methods of dividing the files or packets, the research presented in this thesis provide a foundation for incorporating a dialable security and performance solution with binary integer programming to optimize commodities that can be transmitted over a wireless or wired network.

5.3 Significance of Research

Wireless technology provides the military a direct and immediate communication path between commanders and the battlespace. This path can be between sensors, warfighters with JTRS or "militarized" cell phones or PDAs, unmanned vehicles, highly sensorized aircraft, or via wireless networks.

The technology has inherent vulnerabilities that the military must address to continue to provide confidentiality, integrity, and availability of its resources and to

provide decision makers an intelligent view of the battlespace. To do this, security is paramount in the utilization of wireless technology. Research into securing wireless resources will benefit the military and its operations.

NetD must address the security requirements of wireless devices as our forces become more mobile and more technically advanced.

Using adaptive security, a system can exist in a less secure but higher performing state for normal operations and then can adapt to a more secure and usually less performing state when negative factors arise within or outside the system. Because it is adaptive, the system can utilize different cryptographic algorithms.

This research lays a foundation for adapting the cryptographic algorithms for transmitting packets or files based upon the dynamics within a wireless or wired environment, namely, available bandwidth, available CPU, the particular commodity (size and priority), and the security and performance levels. By creating the controllers to facilitate adaptive cryptography, this thesis provides three methods for selecting encryption schemes for maximizing the number of commodities that can be transmitted from a node.

The first method, Controller One, uses the security levels and performance levels to determine a range of encryption schemes. From this range, an optimal encryption scheme is chosen. Controller Two uses a binary integer solver to determine the optimal encryption scheme while Controller Three combines security levels and performance levels in solving the binary integer problem. These three controllers provide a method to

automate encryption selections (Controller Two), to override a binary solution, and provide more user input (Controller One), or if necessary, combine the selection process.

Appendix 1 GpgTester

The gpgTester was written in C++ by Matt Weeks. Some modifications were made to ensure the output data was consistent with the encryption algorithms. It was used to test different cryptographic algorithms and file sizes by using system calls to the GPG tool. GPG (GNU Privacy Guard) is a free public key cryptographic command line tool for encrypting and decrypting data and for creating digital signatures using the OpenPGP standard defined by RFC 2440. The version used in this research was version 1.4.5 with copywrite 2006 from Free Software Foundation, Inc.

The cryptographic algorithms supported by GPG were:

- Public key: RSA, RSA-E, RSA-S, Elg-E, DSA
- Cipher key: 3DES, CAST5, BLOWFISH, AES, AES 192, AES256, TWOFISH
- Hash: MD5, SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224
- Compression: Uncompressed, ZIP, ZLIB, BZIP2

For this research only RSA, ElGamal, AES, AES 192, 3DES, AES 256, TwoFish, BlowFish, and CAST5 were used along with the various compression options (no hash testing, although gpgTester supported hashes).

The key length (or size) varied for RSA and ElGamal. The RSA key could not go below 1024 while ElGamal could go as low as 768. The default for generating keys is DSA and ElGamal with a default size of 1024. (Kidwell 2005)

GPG stores the public and private key on the computer for public key cryptography. For security, GPG encrypts the private key with a short passphrase (usually good to have a random string of words for the passphrase).

Working within a cygwin environment, gpgTester generated random files for encrypting and decrypting using different algorithms. By trying different sizes with different cryptographic algorithms, the gpgTester was verified to work as required regardless of the algorithm or size file chosen.

To verify that the random files were actually distinct, a separate program, testingDiff.cpp, was created with primarily the same code as gpgTester. TestingDiff.cpp created and saved 30 random files (the same number of files generated by gpgTester). These files were then compared to each other to verify that the files were indeed distinct from each other.

Since gpgTester made system calls to gpg, the algorithms chosen by the user was verified to ensure that the actual algorithm was run via the system calls. To do this verification on the encrypted file, a decryption was necessary. The decryption stated what algorithm was used to encrypt the file.

Appendix 2 List of Computer Code Generated

For this research, the following programs were created:

1. gpgTester – This is the front end for gpg. This was written in C++. It made system calls to gpg with randomly generated files.
2. testingDiff – This code was written in C++ to verify that the random files generated by gpgTester were indeed random.
3. gpgTesterFile – This was the same code as gpgTester but it called gpg with non-random files.
4. inputCreate3DES, inputCreateAES, inputCreateBlowFish, inputCreateRSA, inputCreateElg, inputCreateCAST5, inputCreateTwoFish – These files (all MatLab) read the Results.xls spreadsheet and wrote the output to testing1.M file to create a matrix for the controllers. See Figure 29 for an interaction of these files.
5. inputVariable.m –This file read the data from the matrix testing1.m and output a formatted matrix for other MatLab files.
6. createMatrixA.m – This was used to generate the correct constraints matrix, *A*.
7. indeSecurity.m, indeSecurity3.m– These files were used to determine the encryption scheme based upon the security and performance levels.
8. securityAlg.m – This file was used to collect the seven different encryption algorithms for Controller Two.
9. inputCommodities.m – This file randomly generated the input file size and priority per commodity.

10. runCryptGrav, runCryptGrav2, runCryptGrav3 – These files performed the data interpolation for the given input file size.
11. inputCrypt, inputCrypt2, inputCrypt3 – These were the files which corresponded to Controller One, Two, and Three respectively. They optimized the data from the runCryptGrav files.
12. encryptFitter – This file was used to generate output files for the NS-2 simulations.

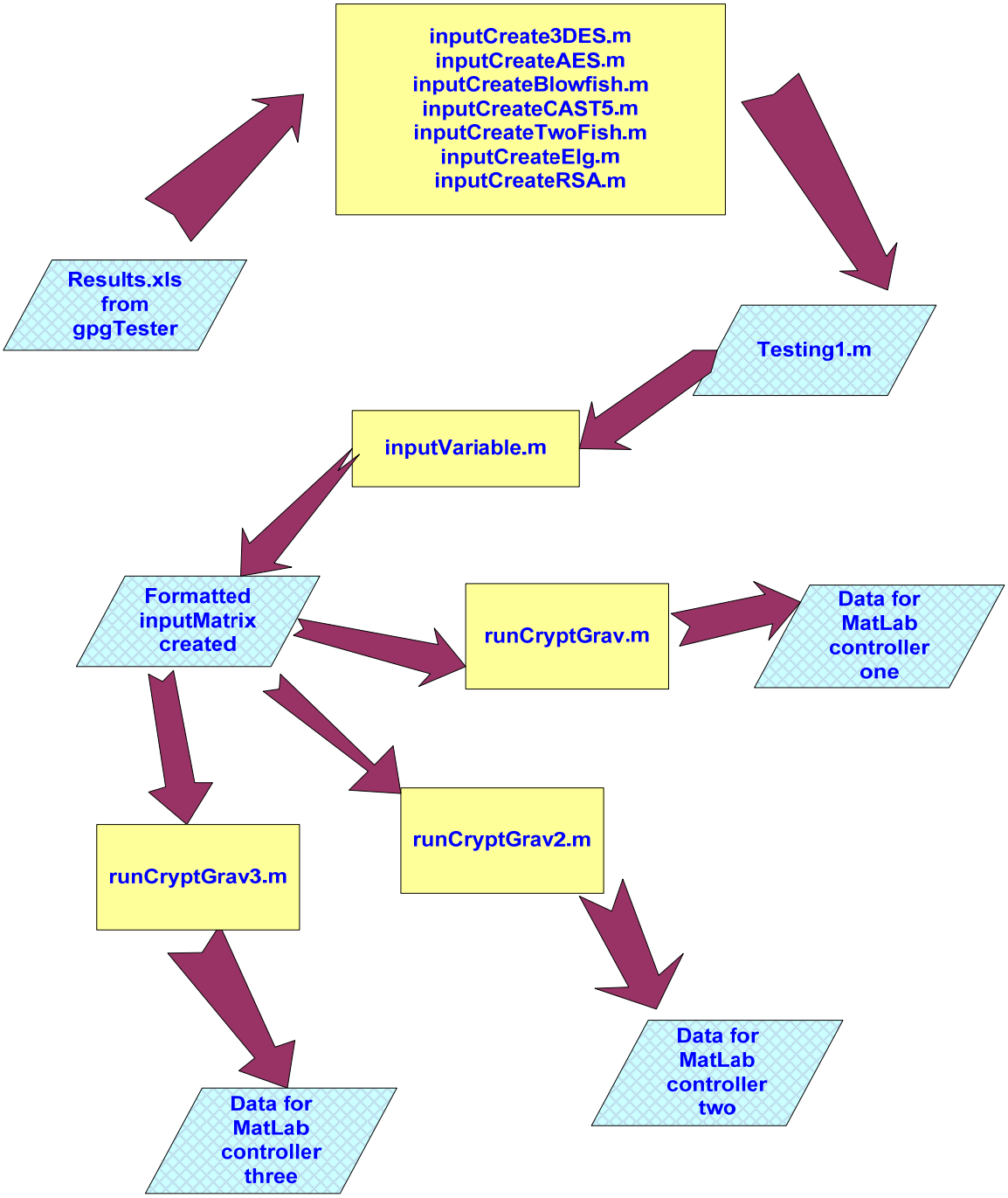

Figure 29. MatLab Files Interaction

Bibliography

1. Alampalayam, S.P. and A. Kumar, An Adaptive Security Model for Mobile Agents in Wireless Networks. Globecom, 2003: p. 6.

2. ANSI Website. [cited 15 Dec 2007]; Available from: http://www.ansi.org/.

3. Bandera, C., et al., Wireless Just-in-Time Training of Mobile Skilled Support Personnel. Proc. of SPIE, 2006. (62500R-1).

4. Basagni, S., et al., Secure Pebblenets. MobiHOC, 2001: p. 8.

5. Bass, M.S.D., The Challenges of Information Management in the Networked Battlespace: Unmanned Aircraft Systems, Raw Data and the Warfighter, in AFIT/ENG. 2006, Air Force Institute of Technology: Wright Patterson Air Force Base. p. 54.

6. Bidigare, P., C. Kreucher, and R. Conti, A Tracking Approach to Localization and Synchronization in Mobile Ad-hoc Sensor Networks. Proc. of SPIE, 2006. (62490l-1).

7. Blyler, J. Wireless Roots Pay Off for Defense Technology. 2004 [cited May 18, 2007]; Wireless technology and how the military has advanced it and now how the commercial market is advancing it]. Available from: http://www.wsdmag.com/Articles/Print.cfm?ArticleID=8360.

8. Brezillon, P. and G.K. Mostefaoui. Context-Based Security Policies: A New Modeling Approach. in Second IEEE Annual Conference on Pervasive Computing and Communications Workshops. 2004.

9. Chigan, C., Y. Ye, and L. Li, Balancing Security against Performance in Wireless Ad Hoc and Sensor Networks. IEEE, 2004: p. 5.

10. Defense, D.o., Use of Commercial Wireless Devices, Services, and Technologies in the Department of Defense (DoD) Global Information Grid (GIG), D.o. Defense, Editor. 2004.

11. Fahey, L.S., Joint Command and Control on the Move (C2OTM). MilCom, 2005. (Joint Systems Integration Command).

12. Franz, M.T.P., Information Operations Foundations to Cyberspace Operations: Analysis, Implementation Concept, and Way-Ahead for Network Warfare Forces. 2007, Air Force Institute of Technology: Wright Patterson Air Force Base. p. 172.

13. Graham, D.F., K.M. Hopkinson, and S.R. Graham, On-Demand Key Distribution for Mobile Ad-Hoc Networks. 2007, Air Force Institute of Technology: Wright Patterson Air Force Base. p. 14.

14. Guenther, R.B. and A. Kharab, An Introduction to Numerical Methods: A MatLab Approach. 2002, New York: Chapman & Hall.

15. Hinton, H., et al., SAM: Security Adaptation Manager, Ryerson Polytechnic University, Canada

16. Kang, K.-D. and S.H. Son. Systematic Security and Timeliness Tradeoffs in Real-Time Embedded Systems. in 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. 2006: IEEE.

17. Kidwell, B. A Practical Introduction to GNU Privacy Guard in Windows. 2005 [cited 2007 March 22]; Available from: http://www.glump.net/dokuwiki/gpg/gpg_intro.

18. Laboratory, A.F.R., AFRL Develops Intrusion Detection and Policy Monitoring for Wireless Networks: Wright Patterson Air Force Base, Ohio.

19. Mais Nijim, X.Q., Tao Xie, Adaptive Quality of Security Control in Networked Parallel Disk Systems.

20. Manzanares, A.I., J.M.S. Camara, and J.T. Marquez, On the Implementation of Security Policies with Adaptative Encryption. ScienceDirect, 2005: p. 9.

21. MatLab Website. [cited 2008 Jan 8, 2008]; Available from: http://www.mathworks.com/products/optimization/description6.html.

22. Melloy, M.J.R., Wireless Sensor Network Applications for the Combat Air Forces, in AFIT Eng. 2006, Air Force Institute of Technology: Wright Patterson Air Force Base. p. 84.

23. Mollin, R.A., Codes: The Guide to Secrecy from Ancient to Modern Times. Discrete Mathematics and Its Applications, ed. K.H. Rosen. 2005,Boca Raton: Chapman and Hall/CRC.

24. Nijim, M., X. Qin, and T. Xie, Adaptive Quality of Security Control in Networked Parallel Disk Systems. IEEE, 2006: p. 6.

25. Pecarina, J.M, Creating an Agent Based Framework to Maximize Information Utility. 2008. Air Force Institute of Technology, Wright Patterson Air Force Base.

26. Pucker, L., Trends in DSP: Can the Military Use Commercial Wireless Signal Processing Technologies to Reduce Size, Weight, and Power in Radio Devices, in IEEE Communications Magazine. 2007.

27. Raissi, J., Dynamic Selection of Optimal Cryptographic Algorithms in Runtime Environment, in 2006 IEEE Congress on Evolutionary Computation. 2006: Sheraton Vancouver Wall Centre Hotel, Vancouvr, British Columbia, Canada.

28. Rhodes, K.A. and G.C. Wilshusen, Information Security: Federal Agencies Need to Improve Controls over Wireless Networks, G.A. Office, Editor. 2005.

29. Sastry, A.R., Autonomous Mobile Mesh Networks and Applications for Defense Network-Centric Operations. Proc. of SPIE, 2006. (62490R-1): p. 10.

30. Soliman, H.S. and M. Omar. Application of Synchronous Dynamic Encryption System in Mobile Wireless Domains. in First ACM (Association for Computer Machinery) International Workshop on Quality of Service and Security in Wireless and Mobile Networks. 2005. Montreal, Quebec, Canada.

31. Spline Website. [cited 15 Dec 2007]; Available from: http://end.wikipedia.org/wiki/Spline_(mathematics).

32. Stamp, M., Information Security: Principles and Practice. 2006, Hoboken, NJ: Wiley and Sons, Inc.

33. Tomlinson, P.G. and J.C. Ricklin, Tactical Optical Systems for a Network-Centric Environment. Proc. of SPIE, 2006. (624909).

34. Yalmip Website. [cited 8 Feb 2008]; Available from: http://control.ee.ethz.ch/~joloef/wiki/pmwiki.php?n=Solvers.BINTPROG.

35. Zou, J., K. Lu, and Z. Jin, Architecture and Fuzzy Adaptive Security Algorithm in Intelligent Firewall. IEEE, 2002(0-7803-7625-0/02): p. 5.

Vita

Major Marnita Thompson Eaddie graduated from Shaw High in East Cleveland, Ohio. She completed her undergraduate studies at Bowdoin College in Brunswick, Maine in 1990 with a degree in Chemical Physics and a minor in Computer Science. She was commissioned in 1997 through the Air Force Officer Training School.

Her first assignment was at Robins Air Force Base, Georgia with the 93rd Air Control Wing, JSTARS in 1997 where she stood up the Information Assurance Office for JSTARS. She later transferred to Kirtland Air Force Base, New Mexico to work with AFOTEC in 2001. Her next assignment (in 2004) was to Ottawa, Ontario where she worked directly with the Canadian Military in delivering a national command and control system to the Canadian Forces. In August 2006, she entered the Air Force Institute of Technology under the Information Assurance Scholarship Program. Upon graduation, she will be assigned to Incirlik Air Base, Turkey.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
27-03-2008	Master's Thesis	October 2006 – March 2008

4. TITLE AND SUBTITLE

DIALABLE CRYPTOGRAPHY FOR WIRELESS NETWORKS

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Marnita Thompson Eaddie, Major, USAF

5d. PROJECT NUMBER
JON # 08-175

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Building 640
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCO/ENG/08-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Robert Bonneau
AFOSR/NL
875 North Randolph Street, Suite 325, Room 3112
Arlington, Virginia 22203 DSN 426-9545

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR/NL

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The objective of this research is to develop an adaptive cryptographic protocol, which allows users to select an optimal cryptographic strength and algorithm based upon the hardware and bandwidth available and allows users to reason about the level of security versus the system throughput. In this constantly technically-improving society, the ability to communicate via wireless technology provides an avenue for delivering information at anytime nearly anywhere. Sensitive or classified information can be transferred wirelessly across unsecured channels by using cryptographic algorithms. The research presented will focus on dynamically selecting optimal cryptographic algorithms and cryptographic strengths based upon the hardware and bandwidth available. The research will explore the performance of transferring information using various cryptographic algorithms and strengths using different CPU and bandwidths on various sized packets or files.

This research will provide a foundation for dynamically selecting cryptographic algorithms and key sizes. The conclusion of the research provides a selection process for users to determine the best cryptographic algorithms and strengths to send desired information without waiting for information security personnel to determine the required method for transferring. This capability will be an important stepping stone towards the military's vision of future Net-Centric Warfare capabilities.

15. SUBJECT TERMS
Adaptive cryptography, binary integer programming, wireless network

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Kenneth M. Hopkinson
U	U	U	UU	125	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-3636, ext 4579 (Kenneth.hopkinson@afit.edu)